Backward under-approximations in numeric abstract domains to automatically infer sufficient program conditions

Antoine Miné

Backward Under-Approximations in Numeric Abstract Domains to Automatically Infer Sufficient Program Conditions[☆,☆☆]

Antoine Miné[*]

CNRS & École normale supérieure, France

Abstract

In this article, we discuss the automatic inference of sufficient preconditions by abstract interpretation and sketch the construction of an under-approximating backward analysis. We focus on numeric properties of variables and revisit three classic numeric abstract domains: intervals, octagons, and polyhedra, with new under-approximating backward transfer functions, including the support for non-deterministic expressions, as well as lower widenings to handle loops. We show that effective under-approximation is possible natively in these domains without necessarily resorting to disjunctive completion nor domain complementation. Applications include the derivation of sufficient conditions for a program to never step outside an envelope of safe states, or dually to force it to eventually fail. We built a proof-of-concept prototype implementation and tried it on simple examples. Our construction and our implementation are very preliminary and mostly untried; our hope is to convince the reader that this constitutes a worthy avenue of research.

Keywords: abstract interpretation, backward analysis, numeric abstract domains, static analysis, under-approximation

1. Introduction

A major problem studied in program verification is the automatic inference of program invariants using forward analyses, as well as the inference by backward analysis of necessary conditions for programs to be correct. In this article, we consider the dual problem: the inference by backward analysis of *sufficient conditions* for programs to be correct.

Motivation. As motivating example, consider the simple loop in Fig. 1.(a): `j` starts from a random value in $[0; 10]$ and is incremented by a random value in $[0; 1]$ at each iteration. A forward invariant analysis would find that, at the end of the loop, $\texttt{j} \in [0; 110]$ and the assertion at line (8) can be violated. We are now interested in inferring conditions on the initial states of the program assuming that the assertion actually holds. A backward analysis of necessary conditions would not infer any new condition on the initial value of `j` because any value in $[0; 10]$ has an execution satisfying the assertion (consider, for instance the executions where the random choice $[0; 1]$ always returns 0). However, a backward sufficient condition analysis would compute the set of initial states such that *all* executions necessarily satisfy the assertion. It will infer the condition: $\texttt{j} \in [0; 5]$ as, even if $[0; 1]$ always evaluate to 1 in the loop, $\texttt{j} <= 105$ holds nevertheless. Necessary and sufficient conditions thus differ in the presence of non-determinism.

[☆]This work is supported by the INRIA project "Abstraction" common to CNRS and ENS in France.

[☆☆]This work is an extended version of the article [1] published in the Proceedings of the 4th International Workshop on Numerical and Symbolic Abstract Domains (NSAD 2012).

[*]Corresponding address: Antoine Miné, Département d'informatique, École normale supérieure, 45 rue d'Ulm, F-75230 Paris Cedex 05, France. Tel.: +33 1 44 32 21 17.

Email address: mine@di.ens.fr (Antoine Miné)

URL: http://www.di.ens.fr/~mine (Antoine Miné)

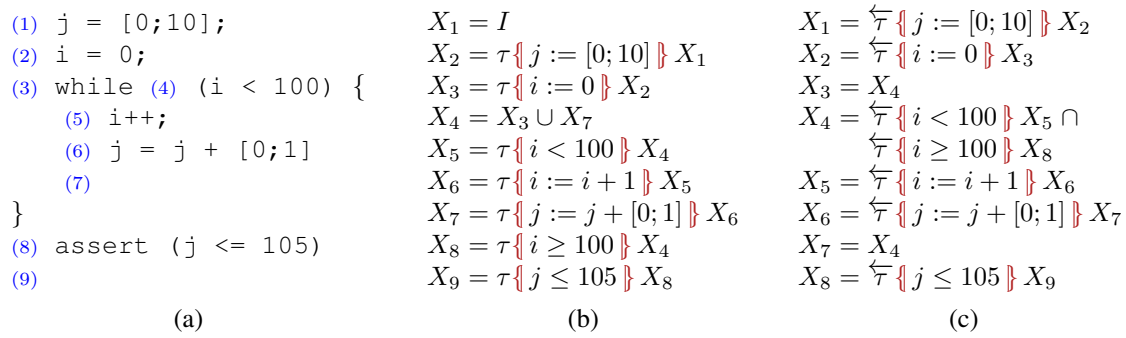

Figure 1: A simple loop program (a), its concrete invariant equation system (b), and its concrete sufficient condition equation system (c).

Sufficient conditions have many applications, including: contract inference, counter-example generation, optimizing compilation, verification driven by temporal properties, etc. Contract inference [2] consists in inferring sufficient conditions at a function entry that ensure that its execution is error-free. Similarly, counter-example generation [3] infers initial states that guarantee that the program goes wrong. Safety checks hoisting, a form of compiler optimization, consists in replacing a set of checks in a code portion (such as a loop or method) with a single check at the code entry. For instance, array bound check hoisting (as done in [4]) infers sufficient conditions under which all array accesses in a method are correct, and then inserts a dynamic test that branches to an optimized, check-free version of the method if the condition holds but reverts to the original method if it does not. A final example use is the verification of temporal properties of programs, such as CTL formulas, as done by Massé [5], where necessary post-conditions as well as necessary and sufficient preconditions are mixed to take into account the interplay of modalities ($\Box$ and $\Diamond$).

Formal methods. Determining the conditions under which a program is correct corresponds to inferring Dijkstra's weakest liberal preconditions [6]. The support for non-deterministic expressions, which was not present in Dijkstra's original presentation, was later added by Morris [7]. Weakest preconditions, and more generally predicate transformers, form the base of current deductive methods (see [8] for a recent introduction). They rely on the use of theorem provers or proof assistants and, ultimately, require help from the user to provide predicates and proof hints. Weakest preconditions also appear in model-checking in the form of modal operators [9]. Many instances of model-checking are based on an exhaustive search in the state space, represented in extension or symbolically, while recent instances build on the improvement in solvers (such as SAT modulo theory [10]) to handle infinite-state spaces. Refinement methods, such as counter-example-guided abstract refinement [11], also merge model-checking with weakest preconditions computed by solvers. In this article, we seek to solve the sufficient condition inference problem using *abstract interpretation* instead. The benefits are: full automation, support for infinite-state systems, parametrization by a choice of domain-aware semantic abstractions and dedicated algorithms, and independence from a generic solver, with the promise to scale up to large programs.

Abstract interpretation. Abstract interpretation [12] is a very general theory of the approximation of program semantics. It stems from the fact that the simplest inference problems on programs are undecidable (or, at least, grow too quickly in cost with the size of the state space to be of any use), and so, approximations are required to achieve scalable and yet fully automatic analyses. It has been applied with some success to the automatic generation of invariants on industrial applications and led to commercial tools, such as Astrée [13]. Its principle is to replace the computations on state sets (so-called concrete semantics) with computations in computer-representable abstractions that, for the sake of efficiency, only represent a selected subset of program properties and ignore others. Numeric abstract domains, which reason on the numerical properties of variables, are widely used and much effort has been spent designing domains adapted to selected properties or achieving a selected cost versus precision trade-off. The two most popular domains are: the interval domain (introduced by Cousot and Cousot in [14]) that infers variable bounds, and the polyhedra domain (introduced by Cousot and Halbwachs in [15]) that infers affine inequalities on variables. A more recent example is the octagon domain [16], which infers unit binary inequalities and thus achieves a balance between intervals and polyhedra in terms of cost and precision. Classic abstract domains enjoy abstract operators for forward and backward analyses, but their backward operators are geared towards the inference of necessary conditions and not the inference

of sufficient ones. Moreover, when an exact result cannot be computed (due to the limited expressiveness of the domain or the impracticability of computing a precise solution), the domains settle for an over-approximation. This is adequate for the inference of invariants and necessary conditions as an over-approximation of the tightest program invariant (resp. the strongest necessary condition) is still an invariant (resp. a necessary condition), but soundness when inferring sufficient conditions requires instead *under-approximations*.

Although under-approximations are theorized in the early works on abstract interpretation by Cousot [12], practical instances are much rarer than over-approximations. Bourdoncle acknowledged the need for under-approximations to perform abstract debugging [17], but did not propose any. We attribute this lack to the perceived difficulty in designing theoretically optimal [18] as well as practical under-approximations in expressive numeric domains (which are infinite), and the fact that sufficient and necessary conditions differ in the presence of non-determinism. One solution [19, 3], limited to the case of deterministic programs (where necessary and sufficient conditions coincide), is to use exact abstractions (e.g., disjunctive completions). Unfortunately, exact abstractions may not scale well. A variant, akin to symbolic execution [20], consists in extracting a finite subset of program paths, which are then tested for satisfiability with an exact method. Hence, only the control space of the program is under-approximated, not its data space. Another solution is to use set-complements of over-approximating domains [21]. Unfortunately, applying this method to classic numeric domains rarely gives interesting invariant forms (for instance, the complement of a polyhedron is a disjunction of affine inequalities, which is ill-equipped to reason about programs with multiple assertions; we will see such an example later in Fig. 11). Another proposal consists in using existentially quantified domains [22, 5, 23], expressing properties such as "there exists a program state satisfying some abstract property." These domains are parametrized by an over-approximating domain (such as intervals in [23]), but inherently express properties of a very different kind. In this article, we seek instead to infer sufficient conditions directly expressible in the chosen abstract domain (not their disjunctive completion, nor complement, nor existentially quantified version); for instance, we use polyhedra to infer quantifier-free conjunctions of inequalities.

Contribution. In this article, we present preliminary work towards the fully automatic analysis of sufficient conditions by abstract interpretation. We introduce backward operators for classic numeric domains: intervals, octagons and polyhedra. These operators correctly handle non-determinism and employ under-approximations to ensure soundness and scalability. Our operators are approximate and, as we refrain from systematically using disjunctive completions, there generally does not exist a best abstraction. We also present the first instance of a lower widening, which allows under-approximating the effect of loops in finite time. The result is a static analysis presentation very similar to the original work of Cousot, Cousot, and Halbwachs [14, 15], but to infer sufficient conditions instead of invariants. We built a proof-of-concept implementation and successfully analyzed a few simple examples similar to Fig. 1 as well as Bubble Sort (Fig. 11) from [15], but we admit that a large amount of work (in the form of new domains and new abstract operators) is required to make the analysis practical and useful.

Outline. The article is organized as follows. Section 2 presents the concrete semantics underlying the computation of sufficient conditions. In particular, it presents very general properties of sufficient condition transfer functions that simplify considerably the design of concrete and abstract semantics. Section 3 then shows the construction of under-approximated sufficient condition operators in some classic numeric domains: intervals, octagons, and polyhedra. Section 4 shows how to extend our analysis to infer inevitability properties and to infer conditions on program environments. Section 5 presents our preliminary prototype implementation and a few example analyses. Section 6 discusses related work and Section 7 concludes. Proofs are relegated to Appendix A. This article extends an article published in the Proceedings of the 4th Workshop on Numerical and Symbolic Abstract Domains [1]. Additional material that did not fit in the workshop version include: a more complete description of the concrete and abstract semantics, the support for intervals and octagons in addition to polyhedra (Sec. 3), a discussion of the integer semantics (Sec. 3.7), inference of conditions on the environment (Sec. 4.2), additional illustrations and examples (Sec. 5), and the proof of all theorems (Appendix A).

2. Concrete semantics

As with all abstract interpretations, we start by defining a concrete semantics, which is the most precise mathematical expression of the program properties we wish to analyze. In addition to formally stating the problem at hand,

the goal of this section is to show how backward sufficient condition operators can be derived systematically from the classic post-condition ones. Approximating this uncomputable semantics with numeric abstractions, which will be considered in Section 3, will be made simpler by applying this observation.

2.1. Invariants and sufficient conditions in transition systems

We first state the problem in its most general form. Following [12], we model programs as transition systems, which are a very general, language-independent form of small-step operational semantics. A transition system (Σ, τ) is composed of a set of states Σ and a transition relation $\tau \subseteq \Sigma \times \Sigma$ modeling atomic execution steps. An execution trace is a finite or infinite countable sequence of states $(\sigma_1, \ldots, \sigma_i, \ldots) \in \Sigma^\infty$ such that $\forall k : (\sigma_k, \sigma_{k+1}) \in \tau$.

Invariant. The invariant inference problem consists in, given a set $I \subseteq \Sigma$ of initial states, inferring the set $\mathrm{inv}(I)$ of states encountered in all executions starting in I. It is actually sufficient to look at all the finite prefixes of (finite or infinite) executions, and gather their final state:

$$\mathrm{inv}(I) \stackrel{\text{def}}{=} \{\, \sigma \mid \exists \sigma_0, \ldots, \sigma_n : \sigma_0 \in I,\ \sigma = \sigma_n,\ \forall i < n : (\sigma_i, \sigma_{i+1}) \in \tau \,\} \ .$$

Following Cousot [12], this set can be expressed in fixpoint form as:

$$\begin{array}{l} \mathrm{inv}(I) = \mathrm{lfp}\, \lambda X. I \cup \mathrm{post}(X) \\ \text{where } \mathrm{post}(X) \stackrel{\text{def}}{=} \{\, \sigma \in \Sigma \mid \exists \sigma' \in X : (\sigma', \sigma) \in \tau \,\} \end{array}$$

where $\mathrm{lfp}\, f$ denotes the least fixpoint of a function f and post is the post-condition operator associating to some states the set of their possible successors. This fixpoint exists according to Tarski [24] as the function is increasing in the complete lattice of state sets $(\mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap)$. In this article, we use the following, alternate form:

$$\mathrm{inv}(I) = \mathrm{lfp}_I\, \lambda X. X \cup \mathrm{post}(X) \tag{1}$$

where $\mathrm{lfp}_x\, f$ is the least fixpoint of f greater than or equal to x. Both forms are equivalent and equal $\bigcup_{n \geq 0} \mathrm{post}^n(I)$ because post is a $\cup$-morphism[1] [25]. However, (1) will prove more convenient as it expresses inv as a fixpoint of a function that does not depend on I.

Sufficient conditions. In this article, we do not consider invariant inference, but the reverse problem: sufficient condition inference. It consists, given an invariant set T to obey, in inferring the set of initial states $\mathrm{cond}(T)$ that guarantee that all (finite and infinite) executions stay in T:

$$\mathrm{cond}(T) \stackrel{\text{def}}{=} \{\, \sigma \mid \forall \sigma_0, \ldots, \sigma_n : \sigma = \sigma_0 \wedge \forall i < n : (\sigma_i, \sigma_{i+1}) \in \tau \implies \forall i \leq n : \sigma_i \in T \,\} \ .$$

It is also given in fixpoint form, following Bourdoncle [17]:

$$\begin{array}{l} \mathrm{cond}(T) = \mathrm{gfp}\, \lambda X. T \cap \widetilde{\mathrm{pre}}(X) \\ \text{where } \widetilde{\mathrm{pre}}(X) \stackrel{\text{def}}{=} \{\, \sigma \in \Sigma \mid \forall \sigma' \in \Sigma : (\sigma, \sigma') \in \tau \Longrightarrow \sigma' \in X \,\} \end{array}$$

where $\mathrm{gfp}\, f$ is the greatest fixpoint of f and $\widetilde{\mathrm{pre}}(X)$ is the set of states such that all their successors by τ are in X. As for inv, we express cond as a fixpoint of a function that does not depend on T:

$$\mathrm{cond}(T) = \mathrm{gfp}_T\, \lambda X. X \cap \widetilde{\mathrm{pre}}(X) \tag{2}$$

where $\mathrm{gfp}_x\, f$ is the greatest fixpoint of f smaller than or equal to x. $\mathrm{cond}(T)$ is indeed a sufficient condition and, in fact, the most general sufficient condition (i.e., the one corresponding to the largest initial state set):

Theorem 1. *$\forall T, X : \mathrm{inv}(\mathrm{cond}(T)) \subseteq T$ and $\mathrm{inv}(X) \subseteq T \Longrightarrow X \subseteq \mathrm{cond}(T)$.*

PROOF. See Appendix A.1. □

[1] A $\cup$- (resp. $\cap$−) morphism f satisfies: for any collection of sets $(X_i)_{i \in I}$, $f(\bigcup_{i \in I} X_i) = \bigcup_{i \in I} f(X_i)$ (resp. $f(\bigcap_{i \in I} X_i) = \bigcap_{i \in I} f(X_i)$). In this article, all morphisms are thus "complete morphisms." Moreover, all $\cup$-morphisms are strict: $f(\emptyset) = \emptyset$.

Non-determinism. The function $\widetilde{\mathrm{pre}}$ we use differs from the function pre used in most backward analyses [17, 12, 26] and defined as $\mathrm{pre}(X) \stackrel{\text{def}}{=} \{\, \sigma \in \Sigma \mid \exists \sigma' \in X : (\sigma, \sigma') \in \tau \,\}$. Indeed, $\widetilde{\mathrm{pre}}(X) \neq \mathrm{pre}(X)$ when the transition system is non-deterministic, i.e., when there exists a state with several successors (e.g., due to a random input) or no successor (e.g., the program is blocked). Non-determinism is extremely useful in practice: it allows modeling unspecified parts of programs (such as unanalyzed libraries) or interactions with the environment (such as the user input in Fig. 1.(a)) as returning any value. Additionally, it is useful to replace complex expressions that cannot be directly handled in an abstract domain with non-deterministic intervals that over-approximate the set of all possible evaluations (see Sec. 3.6). Using $\widetilde{\mathrm{pre}}$ ensures that the target invariant T holds for all the (possibly infinite) sequences of choices made at each execution step; hence, increasing the non-determinism actually reduces the set of predecessors, resulting in an under-approximation. For instance, the program in Fig. 1.(a) can be seen as an abstraction of a family of concrete programs, parameterized by a constant $c \in [0; 1]$, where `j = j + [0;1]` actually over-approximates `j = j +` c. We analyze the abstract program instead of the (possibly infinite) family of concrete ones, confident in the fact that any sufficient condition on the abstract program also holds on each concrete program instance. By contrast, analyses based on pre (such as [27]) infer conditions for the invariant to hold for at least one sequence of non-deterministic choices, but not necessarily all, which is a laxer condition. As a consequence, enlarging the non-determinism also enlarges the image by pre, resulting in an unsound analysis. For instance, an initial value of `j` $= 50$ has an execution satisfying the assertion of the abstract program, although the concrete program instance where $c = 1$ does not satisfy the assertion.

Blocking states. A state σ is blocking if it has no successor. We note B the set of blocking states:

$$B \stackrel{\text{def}}{=} \{\, \sigma \mid \forall \sigma' \in \Sigma : (\sigma, \sigma') \notin \tau \,\} \ .$$

Blocking states denote program termination, either because the program successfully completes, or because it encounters an error — e.g., the statement $y = 1/x$ generates a transition only from states where $x \neq 0$. We note that $\forall X : B \subseteq \widetilde{\mathrm{pre}}(X)$ and, as a consequence, $B \cap T \subseteq \mathrm{cond}(T)$. When computing the predecessors $\widetilde{\mathrm{pre}}(X)$ of some state set X at a given program point, we then must systematically include the blocking states from all program locations, which is rather cumbersome and prevents the construction of the semantics by induction on the syntax. Thus, in the following, we assume that the transition system has no blocking state. Moreover, we wish to distinguish correct program termination from errors (so that we can easily infer sufficient conditions for correct program termination and no error). This can be achieved by adding two special states, α and ω, denoting respectively correct and incorrect program termination, and which are self-loops: $(\alpha, \alpha), (\omega, \omega) \in \tau$. We then complete the transition system by adding transitions from blocking states to either α or ω.

Approximation. Transition systems can become large or infinite, so that neither $\mathrm{inv}(I)$ nor $\mathrm{cond}(T)$ can be computed efficiently or at all. We settle for sound approximations. Invariant sets are over-approximated in order to be certain to include all program behaviors. Dually, sufficient conditions need to be under-approximated. Indeed, inv is a monotonic function [24, 25], and so, any subset $I' \subseteq \mathrm{cond}(T)$ satisfies $\mathrm{inv}(I') \subseteq \mathrm{inv}(\mathrm{cond}(T)) \subseteq T$, i.e., I' is a valid sufficient condition.

Application. Given a set of initial states $I \subseteq \Sigma$ and the error state $\omega \in \Sigma$, we can compute the subset $I_{\overline{\omega}}$ of initial states that never lead to an error as:

$$I_{\overline{\omega}} \stackrel{\text{def}}{=} I \cap \mathrm{cond}(\Sigma \setminus \{\omega\}) \ . \tag{3}$$

An analyzer computing an under-approximation of $I_{\overline{\omega}}$ infers sufficient conditions so that all executions are error-free (i.e., never reach ω). This will be the main kind of properties discussed in this article. We will show in Sec. 4 that it is also possible to infer inevitability properties (i.e., all executions of the program eventually reach a given state set), by reducing them to invariance ones, following the ideas of Cousot and Cousot [28].

2.2. Invariant semantics of numeric programs

Language. We instantiate the preceding framework on properties of numeric programs. For the sake of exposition, we consider a very simple programming language, presented in Fig. 2. The language features a finite, fixed set $\mathcal{V}$ of real-valued variables, simple numeric expressions *expr* and boolean expressions *bexpr*, assignments, tests, and

$$
\begin{array}{lcll}
\textit{prog} & ::= & \textit{stat} & \\
\\
\textit{stat} & ::= & V := \textit{expr} & \wr\ \textit{assignment, } V \in \mathcal{V}\ \wr \\
& | & \texttt{if}\ (\textit{bexpr})\ \{\ \textit{stat}\ \}\ \texttt{else}\ \{\ \textit{stat}\ \} & \wr\ \textit{test}\ \wr \\
& | & \texttt{while}\ (\textit{bexpr})\ \{\ \textit{stat}\ \} & \wr\ \textit{loop}\ \wr \\
& | & \texttt{assert}\ (\textit{bexpr}) & \wr\ \textit{assertion}\ \wr \\
& | & \textit{stat}; \textit{stat} & \wr\ \textit{sequence}\ \wr \\
\\
\textit{expr} & ::= & [a; b] & \wr\ \textit{constant, } a, b \in \mathbb{R} \cup \{-\infty, +\infty\}\ \wr \\
& | & V & \wr\ \textit{variable, } V \in \mathcal{V}\ \wr \\
& | & -\textit{expr} & \wr\ \textit{unary operation}\ \wr \\
& | & \textit{expr} \cdot \textit{expr} & \wr\ \textit{binary operation, } \cdot \in \{+, -, \times, /\}\ \wr \\
\\
\textit{bexpr} & ::= & \textit{expr} \bowtie \textit{expr} & \wr\ \textit{comparison, } \bowtie \in \{<, \leq, >, \geq, =, \neq\}\ \wr \\
& | & \textit{bexpr} \vee \textit{bexpr} & \wr\ \textit{logical or}\ \wr \\
& | & \textit{bexpr} \wedge \textit{bexpr} & \wr\ \textit{logical and}\ \wr \\
& | & \neg \textit{bexpr} & \wr\ \textit{logical negation}\ \wr
\end{array}
$$

Figure 2: Program syntax.

loops. In order to integrate non-determinism directly into the language, we generalize constants in expressions to intervals with constant bounds (each evaluation returns a random number between the bounds). We also assume that statements are tagged with unique program points (sometimes omitted for the sake of concision) ranging in a finite set $\mathcal{L}$. A program state is given by a control location in $\mathcal{L}$ and an environment in $\mathcal{E}$ mapping each variable to its value: $\Sigma \stackrel{\text{def}}{=} \mathcal{L} \times \mathcal{E}$ where $\mathcal{E} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{R}$.

Program semantics are generally not defined as monolithic transition systems, but rather as families of small reusable functions that are composed according to the program syntax. We briefly recall two classic convenient ways to define an invariant semantics: using an equation system and using a big-step semantics; we will present their sufficient condition version in the next section.

Equation systems. A semantics in equational form associates to each control location $\ell \in \mathcal{L}$ a variable X_ℓ with value in $\mathcal{P}(\mathcal{E})$ which denotes the set of possible environments when the program is at control state ℓ. An equation system is derived from the control-flow graph of the program in a mechanical way. This construction is classic and used in many tools (such as the academic analyzer Interproc [29]), so, we only illustrate it on an example: Fig. 1.(b) gives the equation system corresponding to the program in Fig. 1.(a). Given an initial state I specified as the subset of environments $I \subseteq \mathcal{E}$ in which the program starts at its first control location (1), the program semantics is the smallest solution of the equation system. This solution can also be seen as a least fixpoint, and it is the strongest invariant as show by Cousot [30]. The important fact is that the invariant semantics is defined using only three operators: the set union $\cup$, and the semantics of assignments $V := \textit{expr}$ and guards $\textit{bexpr}$, denoted respectively as $\tau\llparenthesis V := \textit{expr} \rrparenthesis$ and $\tau\llparenthesis \textit{bexpr} \rrparenthesis$. The semantic functions $\tau\llparenthesis\ \cdot\ \rrparenthesis$, which we call transfer functions, associate to a set of environments before an instruction the set of environments reachable after the instruction. They are presented in Fig. 3 and use the semantics of expressions $[\![e]\!]\rho$ which returns the set of all possible (numeric or boolean) values e can take in a given environment ρ. In case of a division by zero, expressions evaluate to an error, denoted as ω (for the sake of concision, we omit the propagation of errors ω).

Big-step semantics. Another common presentation of the invariant semantics is as input-output functions (or equivalently, relations) on states. We simply extend the transfer functions of assignments and guards (Fig. 3) to whole programs by induction on the syntax (for the sake of concision, we omit the propagation of errors ω, which are

$$
\begin{array}{l}
\underline{\tau\{\!|\ \cdot\ |\!\} : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E} \cup \{\omega\})} \\
\tau\{\!|\ V := e\ |\!\}\ R \stackrel{\text{def}}{=} \{\, \rho[V \mapsto v] \mid \rho \in R,\, v \in [\![\, e \,]\!]\rho \,\} \cup \{\, \omega \mid \exists \rho \in R : \omega \in [\![\, e \,]\!]\rho \,\} \\
\tau\{\!|\ b\ |\!\}\ R \stackrel{\text{def}}{=} \{\, \rho \mid \rho \in R,\, t \in [\![\, b \,]\!]\rho \,\} \cup \{\, \omega \mid \exists \rho \in R : \omega \in [\![\, b \,]\!]\rho \,\} \\
\\
\underline{[\![\, \mathit{expr} \,]\!] : \mathcal{E} \to \mathcal{P}(\mathbb{R} \cup \{\omega\})} \\
[\![\, [a;b] \,]\!]\rho \stackrel{\text{def}}{=} \{\, x \in \mathbb{R} \mid a \leq x \leq b \,\} \\
[\![\, V \,]\!]\rho \stackrel{\text{def}}{=} \{\rho(V)\} \\
[\![\, -e_1 \,]\!]\rho \stackrel{\text{def}}{=} \{\, -v_1 \mid v_1 \in [\![\, e_1 \,]\!]\rho \,\} \\
[\![\, e_1 \cdot e_2 \,]\!]\rho \stackrel{\text{def}}{=} \{\, v_1 \cdot v_2 \mid v_1 \in [\![\, e_1 \,]\!]\rho,\, v_2 \in [\![\, e_2 \,]\!]\rho,\, v_2 \neq 0 \vee \cdot \neq / \,\} \cup \\
\qquad\qquad \{\, \omega \mid 0 \in [\![\, e_2 \,]\!]\rho \wedge \cdot = / \,\} \\
\\
\underline{[\![\, \mathit{bexpr} \,]\!] : \mathcal{E} \to \mathcal{P}(\{t, f, \omega\})} \\
[\![\, e_1 \bowtie e_2 \,]\!]\rho \stackrel{\text{def}}{=} \{\, t \mid \exists v_1 \in [\![\, e_1 \,]\!]\rho,\, v_2 \in [\![\, e_2 \,]\!]\rho : v_1 \bowtie v_2 \,\} \cup \\
\qquad\qquad \{\, f \mid \exists v_1 \in [\![\, e_1 \,]\!]\rho,\, v_2 \in [\![\, e_2 \,]\!]\rho : v_1 \not\bowtie v_2 \,\} \\
[\![\, b_1 \vee b_2 \,]\!]\rho \stackrel{\text{def}}{=} \{\, t \mid t \in [\![\, b_1 \,]\!]\rho \cup [\![\, b_2 \,]\!]\rho \,\} \cup \{\, f \mid f \in [\![\, b_1 \,]\!]\rho \cap [\![\, b_2 \,]\!]\rho \,\} \\
[\![\, b_1 \wedge b_2 \,]\!]\rho \stackrel{\text{def}}{=} \{\, t \mid t \in [\![\, b_1 \,]\!]\rho \cap [\![\, b_2 \,]\!]\rho \,\} \cup \{\, f \mid f \in [\![\, b_1 \,]\!]\rho \cup [\![\, b_2 \,]\!]\rho \,\} \\
[\![\, \neg b_1 \,]\!]\rho \stackrel{\text{def}}{=} \{\, t \mid f \in [\![\, b_1 \,]\!]\rho \,\} \cup \{\, f \mid t \in [\![\, b_1 \,]\!]\rho \,\}
\end{array}
$$

Figure 3: Invariant semantics of assignments, guards, and expressions.

generated in expressions or failed assertions):

$$
\begin{array}{lcl}
\tau\{\!|\ \texttt{if}\ (b)\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\}\ |\!\}\ R & \stackrel{\text{def}}{=} & (\tau\{\!|\ s_1\ |\!\} \circ \tau\{\!|\ b\ |\!\})R \cup (\tau\{\!|\ s_2\ |\!\} \circ \tau\{\!|\ \neg b\ |\!\})R \\
\tau\{\!|\ \texttt{while}\ (b)\ \{\, s \,\}\ |\!\}\ R & \stackrel{\text{def}}{=} & \tau\{\!|\ \neg b\ |\!\}(\mathrm{lfp}_R\, \lambda X.X \cup (\tau\{\!|\ s\ |\!\} \circ \tau\{\!|\ b\ |\!\})X) \\
\tau\{\!|\ \texttt{assert}\ (b)\ |\!\}\ R & \stackrel{\text{def}}{=} & \tau\{\!|\ b\ |\!\}\ R \cup \{\, \omega \mid \tau\{\!|\ \neg b\ |\!\}\ R \neq \emptyset \,\} \\
\tau\{\!|\ s_1; s_2\ |\!\}\ R & \stackrel{\text{def}}{=} & (\tau\{\!|\ s_2\ |\!\} \circ \tau\{\!|\ s_1\ |\!\})R
\end{array}
\tag{4}
$$

The semantics of a whole program p starting in the initial environment set I is then simply $\tau\{\!|\ p\ |\!\}\ I$, which corresponds to the set of environments reachable at the end of the program. Note that this semantics includes nested fixpoints, to handle nested loops, which are well defined as $\tau\{\!|\ \cdot\ |\!\}$ is a $\cup$-morphism in the complete powerset lattice $\mathcal{P}(\mathcal{E})$. As observed by Schmidt [31], this semantics has a denotational flavor and is quite popular in abstract interpretation (it is used, for instance, in the Astrée analyzer [13]). We also note that this semantics is very similar to an interpreter that executes a program following its syntactic structure forwards from entry to exit and collects the environments encountered. Its benefit over the equational semantics is its parsimonious use of environment sets, leading to memory-efficient analyses which discard invariants at intermediate program points after use. For us, the main point of note is that this semantics remains based on the same atomic operations as the equational one: $\cup$, $\tau\{\!|\ V := \mathit{expr}\ |\!\}$, and $\tau\{\!|\ \mathit{bexpr}\ |\!\}$, with the addition of function composition $\circ$ and nested fixpoints.

2.3. Sufficient condition semantics

We now show how the sufficient condition semantics can be decomposed into a set of atomic transfer functions, similarly to the invariant semantics. More precisely, we present a systematic way to derive sufficient condition transfer functions from invariant ones, and apply it to the invariant semantics of our language.

Backward functions. We start by introducing the backward version, denoted as $\overleftarrow{f}$, of a function $f : \mathcal{P}(X) \to \mathcal{P}(Y)$.

$$
\begin{array}{l}
\overleftarrow{f} : \mathcal{P}(Y) \to \mathcal{P}(X) \\
\overleftarrow{f}(B) \stackrel{\text{def}}{=} \{\, a \in X \mid f(\{a\}) \subseteq B \,\}\ .
\end{array}
\tag{5}
$$

We note immediately that $\overleftarrow{\mathrm{post}} = \widetilde{\mathrm{pre}}$ (see Appendix A.2.7 for a proof). We now state a few interesting algebraic properties of $\overleftarrow{f}$. In the following theorems, we extend $\subseteq$, $\cup$, and $\cap$ element-wise to functions $A \to \mathcal{P}(B)$: we state $f \subseteq g \overset{\text{def}}{\iff} \forall a \in A : f(a) \subseteq g(a)$, $f \cup g \overset{\text{def}}{=} \lambda a.f(a) \cup g(a)$, and $f \cap g \overset{\text{def}}{=} \lambda a.f(a) \cap g(a)$.

Theorem 2.

1. *$\overleftarrow{f}$ is monotonic and a $\cap$-morphism, i.e.,* $\forall (X_i)_{i\in I} : \overleftarrow{f}\left(\bigcap_{i\in I} X_i\right) = \bigcap_{i\in I} \overleftarrow{f}(X_i)$.
2. *$\overleftarrow{f}$ is a sup-$\cup$-morphism, i.e.,* $\forall (X_i)_{i\in I} : \bigcup_{i\in I} \overleftarrow{f}(B_i) \subseteq \overleftarrow{f}\left(\bigcup_{i\in I} B_i\right)$
(in general, it is not a $\cup$-morphism, nor is it strict, even when f is).
3. *If f is monotonic, then $\overleftarrow{f} \circ f$ is extensive, i.e.,* $A \subseteq (\overleftarrow{f} \circ f)(A)$
(in general, the equality does not hold).
4. *If f is a $\cup$-morphism, then $f \circ \overleftarrow{f}$ is reductive, i.e.,* $(f \circ \overleftarrow{f})(B) \subseteq B$
(in general, the equality does not hold).
5. *If f is extensive, then $\overleftarrow{f}$ is reductive; if f is reductive, then $\overleftarrow{f}$ is extensive.*
6. *If f is a $\cup$-morphism, then $\mathcal{P}(X) \underset{f}{\overset{\overleftarrow{f}}{\leftrightarrows}} \mathcal{P}(Y)$ forms a Galois connection,*
i.e., $\forall A \subseteq X, B \subseteq Y : A \subseteq \overleftarrow{f}(B) \iff f(A) \subseteq B$.

PROOF. The proof of these simple properties is presented in Appendix A.2, as well as some counter-examples showing that the equality may not hold in 2–4. □

Property 1 is quite important as it ensures the existence of fixpoints for backward functions. In the, common, case where the forward function is a $\cup$-morphism, property 6 subsumes all the preceding ones.

We continue with a set of properties of the $\overleftarrow{\cdot}$ operator itself, which allow building backward versions of complex functions in terms of backward versions of their constituent:

Theorem 3.

1. $\overleftarrow{\lambda A.A} = \lambda B.B$.
2. $\overleftarrow{f \cup g} = \overleftarrow{f} \cap \overleftarrow{g}$.
3. $\overleftarrow{f \cap g} \supseteq \overleftarrow{f} \cup \overleftarrow{g}$ *(in general, the equality does not hold).*
4. *If f is monotonic, then* $\overleftarrow{f \circ g} \subseteq \overleftarrow{g} \circ \overleftarrow{f}$.
5. *If f is a $\cup$-morphism, then* $\overleftarrow{f \circ g} = \overleftarrow{g} \circ \overleftarrow{f}$.
6. $f \subseteq g \Longrightarrow \overleftarrow{g} \subseteq \overleftarrow{f}$.
7. *If f and g are $\cup$-morphisms, then* $f \subseteq g \iff \overleftarrow{g} \subseteq \overleftarrow{f}$, *and so,* $f = g \iff \overleftarrow{f} = \overleftarrow{g}$.
8. *If f and g are monotonic and $f \circ g = g \circ f = \lambda x.x$, then $\overleftarrow{f} = g$. Moreover, $\overleftarrow{g} = f$ by symmetry.*
9. *If f is a $\cup$-morphism, x is a pre-fixpoint of f, and y is a post-fixpoint of $\overleftarrow{f}$, then* $\mathrm{lfp}_x f \subseteq y \iff x \subseteq \mathrm{gfp}_y \overleftarrow{f}$.
10. *If f is an extensive $\cup$-morphism, then* $\overleftarrow{\lambda x.\mathrm{lfp}_x f} = \lambda y.\mathrm{gfp}_y \overleftarrow{f}$.
11. *If f is a $\cup$-morphism, then* $\overleftarrow{\lambda x.\mathrm{lfp}_x (\lambda z.z \cup f(z))} = \lambda y.\mathrm{gfp}_y(\lambda z.z \cap \overleftarrow{f}(z))$.

PROOF. The proofs are presented in Appendix A.3. □

We now apply these properties to design equational and big-step sufficient condition semantics.

Equation system. The invariant semantics in equational form from Sec. 2.2 has the following general form: $\forall i \in [1;n] : X_i = \bigcup_{j\in[1;n]} F_{i,j}(X_j)$, assuming $\mathcal{L} \overset{\text{def}}{=} [1;n]$, for a family of $\cup$-morphism transfer functions $F_{i,j} : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ and environment set variables $X_1, \ldots, X_n \subseteq \mathcal{E}$. To apply Theorem 3, we rephrase it as a single fixpoint. Given $X \overset{\text{def}}{=} \{\, (i,\sigma) \mid i \in [1;n], \sigma \in X_i \,\}$, we express the program semantics as a function of the initial states $I \subseteq \mathcal{E}$:

$$X = \mathrm{lfp}_{\{1\}\times I} F \text{ where } F(Y) \overset{\text{def}}{=} \bigcup_{i,j} \{\, (i,\sigma) \mid \sigma \in F_{i,j}(\{\sigma'\}), (j,\sigma') \in Y \,\} \ .$$

By applying Theorem 3.2 and Theorem 3.9, we can construct I from a solution of the system X by computing:

$$\mathrm{gfp}_X \overleftarrow{F} \text{ where } \overleftarrow{F}(Y) = \bigcap_{i,j} \{ (j,\sigma) \mid \sigma \in \overleftarrow{F_{i,j}}(\{\sigma'\}), (i,\sigma') \in Y \} \ .$$

Hence, the sufficient condition semantics ensuring that X_i holds at each program point i is the greatest solution smaller than $(X_i)_{i\in[1;n]}$ of the system $\forall j \in [1;n] : Y_j = \bigcap_{i\in[1;n]} \overleftarrow{F}_{i,j}(Y_i)$. For instance, Fig. 1.(c) gives the equation system for the program of Fig. 1.(a), which we derive from the invariant equation system in Fig. 1.(b).

The atomic transfer functions $F_{i,j}$ in the equation system have the form $\tau\{\!|\, V := \mathit{expr} \,|\!\}$ or $\tau\{\!|\, \mathit{bexpr} \,|\!\}$, with semantics defined in Fig. 3. We denote by $\overleftarrow{\tau}\{\!|\, s \,|\!\}$ the backward version of $\tau\{\!|\, s \,|\!\}$, i.e., $\overleftarrow{\tau}\{\!|\, s \,|\!\} \stackrel{\text{def}}{=} \overleftarrow{\tau\{\!|\, s \,|\!\}}$. Applying (5), we get easily:

$$\begin{array}{lcl} \overleftarrow{\tau}\{\!|\, V := e \,|\!\}\, R & = & \{ \rho \in \mathcal{E} \mid \forall v \in [\![e]\!]\rho : \rho[V \mapsto v] \in R \} \\ \overleftarrow{\tau}\{\!|\, b \,|\!\}\, R & = & R \cup \{ \rho \in \mathcal{E} \mid [\![b]\!]\, \rho = \{f\} \} \end{array} \tag{6}$$

which completes the constructions of the backward semantics in equational form. For the sake of presentation, we have implicitly assumed that R does not contain the error state ω, so that $\overleftarrow{\tau}\{\!|\, s \,|\!\}$ does not contain states that may lead to an error when evaluating $[\![e]\!]\rho$ — this is, for instance, the case when computing $I_{\overline{\omega}}$ (3).

Big-step semantics. Computing sufficient conditions in big-step form is even simpler. We can compute $\overleftarrow{\tau}\{\!|\, s \,|\!\}$ by structural induction on the syntax of the program by noting that the $\cup$, $\circ$, and lfp_x operators used in the definition of $\tau\{\!|\, s \,|\!\}$ (4) can be handled respectively by Theorem 3.2, Theorem 3.5, and Theorem 3.11, which gives:

$$\begin{array}{lcl} \overleftarrow{\tau}\{\!|\, \texttt{if}\ (b)\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\} \,|\!\}\, R & = & (\overleftarrow{\tau}\{\!|\, b \,|\!\} \circ \overleftarrow{\tau}\{\!|\, s_1 \,|\!\})R \cap (\overleftarrow{\tau}\{\!|\, \neg b \,|\!\} \circ \overleftarrow{\tau}\{\!|\, s_2 \,|\!\})R \\ \overleftarrow{\tau}\{\!|\, \texttt{while}\ (b)\ \{\, s \,\} \,|\!\}\, R & = & \mathrm{gfp}_{\overleftarrow{\tau}\{\!|\, \neg b \,|\!\}\, R}\, \lambda X.X \cap (\overleftarrow{\tau}\{\!|\, b \,|\!\} \circ \overleftarrow{\tau}\{\!|\, s \,|\!\})X \\ \overleftarrow{\tau}\{\!|\, \texttt{assert}\ (b) \,|\!\}\, R & = & \overleftarrow{\tau}\{\!|\, b \,|\!\}\, R \cap \overleftarrow{\tau}\{\!|\, \neg b \,|\!\}\, \emptyset \\ \overleftarrow{\tau}\{\!|\, s_1; s_2 \,|\!\}\, R & = & (\overleftarrow{\tau}\{\!|\, s_1 \,|\!\} \circ \overleftarrow{\tau}\{\!|\, s_2 \,|\!\})R \end{array} \tag{7}$$

Again, we implicitly assume that $\omega \notin R$, so that $\overleftarrow{\tau}\{\!|\, s \,|\!\}\, R$ contains only states that cannot result in a division by zero nor an assertion failure (hence the intersection with $\overleftarrow{\tau}\{\!|\, \neg b \,|\!\}\, \emptyset$ in the semantics of $\texttt{assert}\ (b)$). Given a set of target environments $O \subseteq \mathcal{E}$ at the end of the program, $\overleftarrow{\tau}\{\!|\, s \,|\!\}\, O$ gives the set I of initial environments such that all program executions from I either reach the end of the program in a state in O, or never terminate, without ever encountering an error (Sec. 4 will discuss how to additionally enforce termination).

3. Abstract semantics

The concrete semantics is not computable as it manipulates sets in $\mathcal{P}(\mathcal{E}) = \mathcal{P}(\mathcal{V} \to \mathbb{R})$. Even if we replace $\mathbb{R}$ with a more realistic, finite data-type (such as machine integers of floating-point numbers) and the semantics becomes computable, it remains nevertheless impractically large. Abstract interpretation [12] proposes to solve this problem by reasoning on abstract properties instead of concrete sets. One defines a set $\mathcal{D}^\sharp$ of computer-representable properties, so-called abstract elements, together with a data-structure encoding and a partial order $\sqsubseteq^\sharp$ denoting relative precision. A concretization function $\gamma : \mathcal{D}^\sharp \to \mathcal{P}(\mathcal{E})$ associates a meaning to each abstract element as the set of environments it represents, and must be compatible with the precision order (i.e., γ is monotonic). In particular, $\mathcal{D}^\sharp$ features a smallest and a greatest element, $\bot^\sharp$ and $\top^\sharp$, that satisfy $\gamma(\bot^\sharp) = \emptyset$ and $\gamma(\top^\sharp) = \mathcal{E}$.

Abstract invariants. An abstract domain comes with abstract versions $F^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ of all the operators $F : \mathcal{P}(\mathcal{E}) \to \mathcal{P}(\mathcal{E})$ defining the concrete semantics, together with effective algorithms to implement these abstract versions. More precisely, to compute invariants, it is necessary to provide abstract versions $\cup^\sharp$, $\tau^\sharp\{\!|\, V := \mathit{expr} \,|\!\}$, and $\tau^\sharp\{\!|\, \mathit{bexpr} \,|\!\}$ of $\cup$, $\tau\{\!|\, V := \mathit{expr} \,|\!\}$, and $\tau\{\!|\, \mathit{bexpr} \,|\!\}$. They should obey the following soundness condition denoting an over-approximation:

$$\forall X^\sharp \in \mathcal{D}^\sharp : (F \circ \gamma)(X^\sharp) \subseteq (\gamma \circ F^\sharp)(X^\sharp) \ . \tag{8}$$

Additionally, in order to approximate fixpoint computations in finite time, standard static analyses define convergence acceleration operators $\triangledown : \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ called widenings. A widening $\triangledown$ should obey the following soundness and termination conditions:

1. $\forall X^\sharp, Y^\sharp : \gamma(X^\sharp) \cup \gamma(Y^\sharp) \subseteq \gamma(X^\sharp \triangledown Y^\sharp)$;
2. for any sequence $(X_i^\sharp)_{i\in I}$, the sequence defined as $Y_0^\sharp \stackrel{\text{def}}{=} X_0^\sharp$ and $Y_{i+1}^\sharp \stackrel{\text{def}}{=} Y_i^\sharp \triangledown X_{i+1}^\sharp$ reaches a stable iterate $Y_{\delta+1}^\sharp = Y_\delta^\sharp$.

As a result, fixpoints can be approximated in finite time through sequences of the form $X_{i+1}^\sharp \stackrel{\text{def}}{=} X_i^\sharp \triangledown F^\sharp(X_i^\sharp)$.

These operators are sufficient to construct a computable abstract version of either the equational or the big-step semantics; each abstract semantics outputs an abstract invariant over-approximating the optimal invariant computed by the concrete semantics. Whenever an equality holds for (8), the abstract operator is said to be exact, which is rather rare. Abstract domains may enjoy an abstraction function $\alpha : \mathcal{P}(\mathcal{E}) \to \mathcal{D}^\sharp$ such that $\mathcal{P}(\mathcal{E}) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathcal{D}^\sharp$ forms a Galois connection. In this case, every concrete element $X \subseteq \mathcal{E}$ has an optimal abstraction $\alpha(X) \in \mathcal{D}^\sharp$, and every concrete operator F has an optimal abstraction $\alpha \circ F \circ \gamma$. This is quite a desirable property, but by no way mandatory. Some domains do enjoy it (such as intervals and octagons), but other, useful domains do not (such as polyhedra).

Abstract sufficient conditions. As we are interested in under-approximating sufficient conditions, we require instead abstract versions $\cap^\sharp$, $\overleftarrow{\tau}^\sharp\{\!|\, V := expr \,|\!\}$ and $\overleftarrow{\tau}^\sharp\{\!|\, bexpr \,|\!\}$ of the backward operators $\cap$, $\overleftarrow{\tau}\{\!|\, V := expr \,|\!\}$, and $\overleftarrow{\tau}\{\!|\, bexpr \,|\!\}$, obeying the following soundness condition:

$$\forall X^\sharp \in \mathcal{D}^\sharp : (\gamma \circ F^\sharp)(X^\sharp) \subseteq (F \circ \gamma)(X^\sharp) \ . \tag{9}$$

We also require a lower widening $\underline{\triangledown}$ obeying the new soundness condition and the same termination condition as $\triangledown$:

Definition 1.

1. $\forall X^\sharp, Y^\sharp : \gamma(X^\sharp \mathbin{\underline{\triangledown}} Y^\sharp) \subseteq \gamma(X^\sharp) \cap \gamma(Y^\sharp)$;
2. *for any sequence* $(X_i^\sharp)_{i\in I}$, *the sequence defined as* $Y_0^\sharp \stackrel{\text{def}}{=} X_0^\sharp$ *and* $Y_{i+1}^\sharp \stackrel{\text{def}}{=} Y_i^\sharp \mathbin{\underline{\triangledown}} X_{i+1}^\sharp$ *reaches a stable iterate* $Y_{\delta+1}^\sharp = Y_\delta^\sharp$.

Given under-approximations $F^\sharp$ and $X^\sharp$ of a concrete operator F and a concrete element X, the sequence $Y_0^\sharp \stackrel{\text{def}}{=} X^\sharp$, $Y_{i+1}^\sharp = Y_i^\sharp \mathbin{\underline{\triangledown}} F^\sharp(Y_i^\sharp)$ reaches a stable iterate, which we denote as $\lim_{X^\sharp} \lambda Y^\sharp.Y^\sharp \mathbin{\underline{\triangledown}} F^\sharp(Y^\sharp)$, and which is a sound under-approximation of $\mathrm{gfp}_X\, F$:

Theorem 4. $\gamma(\lim_{X^\sharp} \lambda Y^\sharp.Y^\sharp \mathbin{\underline{\triangledown}} F^\sharp(Y^\sharp)) \subseteq \mathrm{gfp}_X\, F$.

PROOF. See Appendix A.4 □

An abstract version of the sufficient condition semantics in big-step form (7) can then be constructed by induction on the syntax:

$$\begin{array}{lcl}
\overleftarrow{\tau}^\sharp\{\!|\, \texttt{if}\ (b)\ \{\, s_1 \,\}\ \texttt{else}\ \{\, s_2 \,\} \,|\!\}\ R^\sharp & \stackrel{\text{def}}{=} & (\overleftarrow{\tau}^\sharp\{\!|\, b \,|\!\} \circ \overleftarrow{\tau}^\sharp\{\!|\, s_1 \,|\!\})R^\sharp \cap^\sharp (\overleftarrow{\tau}^\sharp\{\!|\, \neg b \,|\!\} \circ \overleftarrow{\tau}^\sharp\{\!|\, s_2 \,|\!\})R^\sharp \\
\overleftarrow{\tau}^\sharp\{\!|\, \texttt{while}\ (b)\ \{\, s \,\} \,|\!\}\ R^\sharp & \stackrel{\text{def}}{=} & \lim_{\overleftarrow{\tau}^\sharp\{\!|\, \neg b \,|\!\}\ R^\sharp} \lambda X^\sharp.X^\sharp \mathbin{\underline{\triangledown}} (X^\sharp \cap^\sharp (\overleftarrow{\tau}^\sharp\{\!|\, b \,|\!\} \circ \overleftarrow{\tau}^\sharp\{\!|\, s \,|\!\})X^\sharp) \\
\overleftarrow{\tau}^\sharp\{\!|\, \texttt{assert}\ (b) \,|\!\}\ R^\sharp & \stackrel{\text{def}}{=} & \overleftarrow{\tau}^\sharp\{\!|\, b \,|\!\}\ R^\sharp \cap^\sharp \overleftarrow{\tau}^\sharp\{\!|\, \neg b \,|\!\}\ \bot^\sharp \\
\overleftarrow{\tau}^\sharp\{\!|\, s_1; s_2 \,|\!\}\ R^\sharp & \stackrel{\text{def}}{=} & (\overleftarrow{\tau}^\sharp\{\!|\, s_1 \,|\!\} \circ \overleftarrow{\tau}^\sharp\{\!|\, s_2 \,|\!\})R^\sharp
\end{array} \tag{10}$$

and indeed yields a sound under-approximation:

Theorem 5. *For any program* p *and* $X^\sharp \in \mathcal{D}^\sharp$, $\gamma(\overleftarrow{\tau}^\sharp\{\!|\, p \,|\!\}\ X^\sharp) \subseteq \overleftarrow{\tau}\{\!|\, p \,|\!\}\ \gamma(X^\sharp)$.

PROOF. The proof is immediate by induction on the syntax, by exploiting Theorem 4, the soundness of $\overleftarrow{\tau}^\sharp\{\!|\, V := expr \,|\!\}$, $\overleftarrow{\tau}^\sharp\{\!|\, bexpr \,|\!\}$, $\cap^\sharp$, the monotony of γ, and the monotony of the concrete backward functions (Theorem 2.1). □

Likewise, an abstract equation system can be constructed by replacing equations $Y_j = \bigcap_i \overleftarrow{F}_{i,j}(Y_i)$ with $Y_j^\sharp = \bigcap_i^\sharp \overleftarrow{F}_{i,j}^\sharp(Y_i^\sharp)$. It can then be solved in a classic way, as abstract invariant equation systems are, by iterating $\forall j, n : Y_{j,n+1}^\sharp = \bigcap_i^\sharp \overleftarrow{F}_{i,j}^\sharp(Y_{i,n}^\sharp)$ from $\forall j : Y_{j,0}^\sharp = \top^\sharp$. To ensure convergence, a widening is inserted, replacing $Y_{j,n+1}^\sharp = \bigcap_i^\sharp \overleftarrow{F}_{i,j}^\sharp(Y_{i,n}^\sharp)$ with $Y_{j,n+1}^\sharp = Y_{j,n}^\sharp \mathbin{\underline{\triangledown}} \bigcap_i^\sharp \overleftarrow{F}_{i,j}^\sharp(Y_{i,n}^\sharp)$ at selected locations i such that every variable dependency cycle crosses a widening point (for instance, at point 4 in Fig. 1.(c)). We refer the reader to [32] for more information on how to compute approximate solutions of abstract equation systems.

Abstract operator design. We reduced our sufficient condition analysis to the design of four abstract operators: $\overleftarrow{\tau}^\sharp\{\!|\, V := \mathit{expr} \,|\!\}$, $\overleftarrow{\tau}^\sharp\{\!|\, \mathit{bexpr} \,|\!\}$, $\cap^\sharp$, and $\underline{\triangledown}$. However, unlike over-approximations, most classic domains do not enjoy a best under-approximation of transfer functions; in fact, neither intervals, octagons nor polyhedra do. Research on the use of Galois connections to define a notion of optimal under-approximation, such as the work by Schmidt [18] or by Massé [5], points toward higher-order constructions, such as disjunctive completions or closure operators, which incur a high extra cost. We choose another route, which is to keep abstract domains as-is and design under-approximating operators that are generally non-optimal.

Generally, abstract operators are much easier to define when the involved expressions match the proprieties directly expressible in $\mathcal{D}^\sharp$ (for instance, affine expressions in polyhedra). Hence, it is useful to define generic fallback operators for the case where it is too difficult to come up with a smart operator. For under-approximations, it is always sound to return $\bot^\sharp$, the same way over-approximating analyzers soundly bailout with $\top^\sharp$ in case of time-out or unimplemented operation. Because backward operators are generally not strict (i.e., $\overleftarrow{f}(\emptyset) \neq \emptyset$, as the guard transfer function in Sec. 3.2 will show), returning $\bot^\sharp$ at some point does not prevent finding a non-empty sufficient condition at the entry point; it only means that the analysis forces some program branch to be dead.

3.1. Numeric abstract domains

We are interested in abstracting subsets of $\mathcal{V} \to \mathbb{R}$. We can view a numeric property as a subset of the vector space $\mathbb{R}^n$ where $n = |\mathcal{V}|$, and thus will use standard vector notations: $\vec{a}$ denotes a vector in $\mathbb{R}^n$, $\cdot$ denotes the dot product, and $\vec{\rho} \in \mathbb{R}^n$ denotes the vector of variable values corresponding to an environment $\rho \in \mathcal{V} \to \mathbb{R}$.

Many numeric abstract domains have been proposed. An important instance we consider here is the polyhedra domain; it was introduced in [15] to infer conjunctions of affine inequalities on variables. A (possibly unbounded) polyhedron P can be defined as a set $C = \{\, c_1, \ldots, c_m \,\}$ of affine constraints $c_i = (\vec{a_i} \cdot \vec{x} \geq b_i)$, which represents:

$$\gamma_c(C) \stackrel{\text{def}}{=} \{\, \rho \in \mathcal{E} \mid \forall (\vec{a} \cdot \vec{x} \geq b) \in C : \vec{a} \cdot \vec{\rho} \geq b \,\} \ .$$

Alternatively, a polyhedron can be defined as a set (V, R) of vertices V and of rays R, collectively known as generators, which represents:

$$\gamma_g(V, R) \stackrel{\text{def}}{=} \{\, \sum_{\vec{v} \in V} \alpha_{\vec{v}} \vec{v} + \sum_{\vec{r} \in R} \beta_{\vec{r}} \vec{r} \mid \forall \vec{v} \in V, \vec{r} \in R : \alpha_{\vec{v}}, \beta_{\vec{r}} \geq 0,\ \sum_{\vec{v} \in V} \alpha_{\vec{v}} = 1 \,\} \ .$$

In order to ensure an effective representation in computers, the domain generally encodes coefficients as rationals in $\mathbb{Q}$ and manipulates them using arbitrary precision libraries.

Additionally, we will discuss two important restrictions of polyhedra: intervals and octagons. The interval domain, introduced in [14], can infer variable bounds: $X \in [a; b]$. It is a simple and efficient domain, yet the properties it infers are very useful (for instance, to prove the absence of integer or array index overflow). The octagon domain, introduced more recently [16], is a restriction of polyhedra to unit binary inequality constraints: $\pm V_1 \pm V_2 \leq c$. It subsumes intervals but can also represent a limited class of relations between variables.

These three domains are similar semantically in that they represent convex sets and infer conjunctions of inequalities, but they are based on different algorithms and achieve different trade-offs between precision and efficiency. Interval operators have a linear cost in the number of variables, while octagon ones have a cubic cost. The cost of polyhedra is unbounded (as it can construct arbitrarily many constraints), but it is exponential in practice [33].

An important note is that, as these domains represent conjunctions of constraints, abstract elements are closed under intersection. Thus $\cap^\sharp$ is always an exact abstraction.

3.2. Guards

Fallback operator. We first note that, for any test *bexpr*, we have $\tau\{\!|\, \mathit{bexpr} \,|\!\} \subseteq \lambda R.R$. Thus, applying Theorem 3, we get $\overleftarrow{\tau}\{\!|\, \mathit{bexpr} \,|\!\} \supseteq \overleftarrow{\lambda R.R} = \lambda R.R$. Hence, we can simply abstract $\overleftarrow{\tau}\{\!|\, \mathit{bexpr} \,|\!\}$ as $\overleftarrow{\tau}^\sharp\{\!|\, \mathit{bexpr} \,|\!\} \stackrel{\text{def}}{=} \lambda X^\sharp.X^\sharp$, which is always sound, but coarse. In the following we show how to construct more precise abstractions for selected expressions *bexpr*.

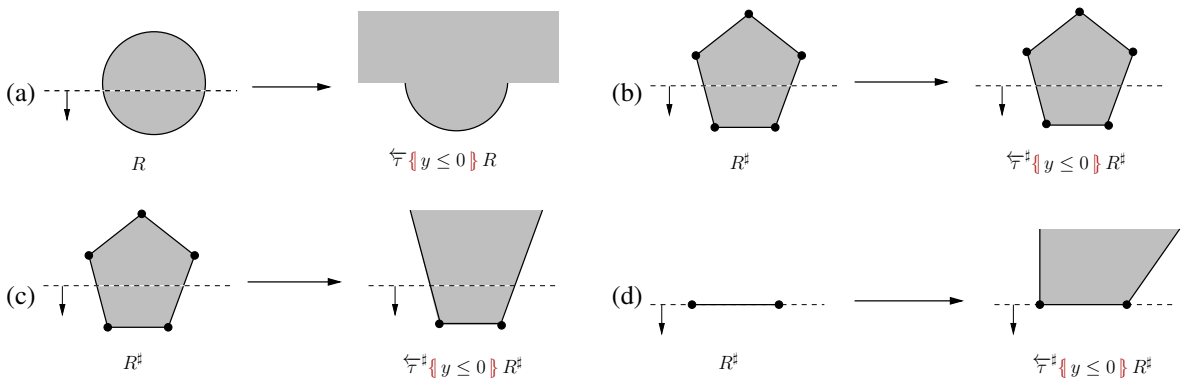

Figure 4: Modeling the guard $y \leq 0$ backward in the concrete (a) and with polyhedra (b)–(d).

Affine guards. We first consider polyhedra, and naturally consider guards of the form $\vec{a} \cdot \vec{x} \geq b$ as they can be modeled exactly by polyhedra. Recall that the concrete forward semantics of a guard is:

$$\tau\{\!|\, \vec{a} \cdot \vec{x} \geq b \,|\!\}\, R \stackrel{\text{def}}{=} \{\, \rho \in R \mid \vec{a} \cdot \vec{\rho} \geq b \,\}$$

and so, the concrete backward semantics is:

$$\overleftarrow{\tau}\{\!|\, \vec{a} \cdot \vec{x} \geq b \,|\!\}\, R = R \cup \{\, \rho \in \mathcal{E} \mid \vec{a} \cdot \vec{\rho} < b \,\} \ . \tag{11}$$

The forward guard on polyhedra simply corresponds to adding the constraint $\vec{a} \cdot \vec{x} \geq b$ to the constraint representation C, which yields an exact abstraction. There is no exact abstraction for the backward guard, however. Indeed, the result of a backward affine guard on a closed convex set is generally not closed nor convex, as shown in Fig. 4.(a). Our first idea is to simply remove $\vec{a} \cdot \vec{x} \geq b$ syntactically from the set of constraints C, which is sound as it only adds points that satisfy $\vec{a} \cdot \vec{x} < b$:

Theorem 6. $\overleftarrow{\tau}\{\!|\, \vec{a} \cdot \vec{x} \geq b \,|\!\}\, \gamma_c(C) \supseteq \gamma_c(C \setminus \{\vec{a} \cdot \vec{x} \geq b\})$.

PROOF. See Appendix A.5. □

Figure 4.(b) shows that this abstract operator can be quite imprecise: when $(\vec{a} \cdot \vec{x} \geq b) \notin C$, it reverts to the identity. Hence, we refine our idea by not only removing $\vec{a} \cdot \vec{x} \geq b$ but also all constraints that are redundant in $C \cup \{\vec{a} \cdot \vec{x} \geq b\}$. Intuitively, these are constraints that restrict $\gamma_c(C)$ in the half-space $\vec{a} \cdot \vec{x} < b$, while the guard result is not restricted in this half-space. An example is shown in Fig. 4.(c). In practice, we first add $\vec{a} \cdot \vec{x} \geq b$, then remove redundant constraints, then remove $\vec{a} \cdot \vec{x} \geq b$. Algorithms to perform redundancy removal are directly available in polyhedra (either using Cherkikova's algorithm or the Simplex algorithm), and so, this operator can be easily implemented.

As a box (i.e., a Cartesian products of intervals) is a special case of polyhedron, we can apply the same idea to the interval domain. Given a set of constraints of the form $\{\, \pm X \leq c \mid X \in \mathcal{V} \,\} \cup \{\vec{a} \cdot \vec{x} \geq b\}$, detecting redundant unary constraints can be performed in quadratic time in the worst case. We can also apply the same technique to octagons. A cubic-time algorithm for detecting redundant constraints has been proposed in [34], resulting in an efficient implementation of $\overleftarrow{\tau}^\sharp\{\!|\, \pm X_i \pm X_j \geq c \,|\!\}$. When the added constraint is not octagonal, it is always possible to use the general Simplex algorithm (the author is unaware of the existence of specialized Simplex algorithms for octagons).

Extended affine guards. We now consider a wider range of affine guards, including strict guards and guards with a non-deterministic constant. We use the following theorem which builds on the results of Theorem 6:

Theorem 7.

1. $\overleftarrow{\tau}\{\!|\, \vec{a} \cdot \vec{x} > b \,|\!\}\, \gamma_c(C) \supseteq \gamma_c(C \setminus \{\vec{a} \cdot \vec{x} \geq b\})$.

2. $\overleftarrow{\tau}\{\!|\, \vec{a}\cdot\vec{x} \geq [b;c] \,|\!\}\, \gamma_c(C) \supseteq \gamma_c(C \setminus \{\vec{a}\cdot\vec{x} \geq b\})$.
3. $\overleftarrow{\tau}\{\!|\, \vec{a}\cdot\vec{x} = [b;c] \,|\!\}\, \gamma_c(C) \supseteq \gamma_c(C \setminus \{\vec{a}\cdot\vec{x} \geq b, (-\vec{a})\cdot\vec{x} \geq (-c)\})$.

PROOF. This is a simple consequence of Theorem 3 and Theorem 6. See Appendix A.6. □

As we will see in Sec. 3.6, guards involving non-linear expressions can be abstracted into affine guards with an interval constant, which can then be handled by the above theorem.

Boolean operations. We now consider boolean conjunctions and disjunctions of affine guards (there is no need to consider boolean negation, which can be eliminated by applying De Morgan's law). Their concrete forward semantics is:

$$\begin{aligned}\tau\{\!|\, t_1 \vee t_2 \,|\!\} &= \tau\{\!|\, t_1 \,|\!\} \cup \tau\{\!|\, t_2 \,|\!\} \\ \tau\{\!|\, t_1 \wedge t_2 \,|\!\} &= \tau\{\!|\, t_2 \,|\!\} \circ \tau\{\!|\, t_1 \,|\!\} \quad .\end{aligned}$$

By applying Theorem 2, we get the following concrete backward semantics:

$$\begin{aligned}\overleftarrow{\tau}\{\!|\, t_1 \vee t_2 \,|\!\} &= \overleftarrow{\tau}\{\!|\, t_1 \,|\!\} \cap \overleftarrow{\tau}\{\!|\, t_2 \,|\!\} \\ \overleftarrow{\tau}\{\!|\, t_1 \wedge t_2 \,|\!\} &= \overleftarrow{\tau}\{\!|\, t_1 \,|\!\} \circ \overleftarrow{\tau}\{\!|\, t_2 \,|\!\} \quad .\end{aligned}$$

Hence, boolean guards can be abstracted using the exact intersection $\cap^\sharp$ and the composition $\circ$ of affine guards. Note that we refrained from defining $\tau\{\!|\, t_1 \wedge t_2 \,|\!\}$ (equivalently) as $\tau\{\!|\, t_1 \,|\!\} \cap \tau\{\!|\, t_2 \,|\!\}$ as applying Theorem 3 would only give an inclusion $\overleftarrow{\tau}\{\!|\, t_1 \wedge t_2 \,|\!\} \supseteq \overleftarrow{\tau}\{\!|\, t_1 \,|\!\} \cup \overleftarrow{\tau}\{\!|\, t_2 \,|\!\}$, which is sound but results in a loss of precision in case the equality does not hold; it would also require under-approximating $\cup$ in our abstract domain, which we try to avoid.

If-then-else. Generally, guards do not appear alone in the program semantics but rather occur as pairs of opposing guards in `if` (b) $\{\, s_1 \,\}$ `else` $\{\, s_2 \,\}$ statements. For instance, in the big-step semantics (10), a test has the form $\overleftarrow{\tau}^\sharp\{\!|\, b \,|\!\}\, Y^\sharp \cap^\sharp \overleftarrow{\tau}^\sharp\{\!|\, \neg b \,|\!\}\, Z^\sharp$. A particularly interesting case occurs when the test is non-deterministic, as both branches may be taken. This is illustrated in Fig. 5 with the guard $y + [0;1] \geq 0$. This test reduces to $y \geq -1$ for the then branch, and $y < 0$ for the else branch. In the forward semantics (Fig. 5.(a), left to right), both branches have environments satisfying $y \in [-1;0]$. In the backward semantics (Fig. 5.(b), right to left), environments satisfying $y \in [-1;0]$ in both branches must match ($\cap$) to be kept in the sufficient condition, i.e., a sufficient condition that may pass both b and $\neg b$ must be a sufficient condition both for the then branch and for the else branch.

Implementing binary transfer functions of the form $\lambda(Y^\sharp, Z^\sharp).\, \overleftarrow{\tau}^\sharp\{\!|\, b \,|\!\}\, Y^\sharp \cap^\sharp \overleftarrow{\tau}^\sharp\{\!|\, \neg b \,|\!\}\, Z^\sharp$ is also useful to improve the precision of abstract guards. Consider, for instance, a degenerate case of a guard $\overleftarrow{\tau}^\sharp\{\!|\, \vec{a}\cdot\vec{x} \geq b \,|\!\}\, \gamma_c(C)$ in a polyhedron such that $\gamma_c(C) \models \vec{a}\cdot\vec{x} = b$, as in Fig. 4.(d). Constraint representations of degenerate polyhedra are not unique, and different choices may result in different outcomes for $\overleftarrow{\tau}^\sharp\{\!|\, \vec{a}\cdot\vec{x} \geq b \,|\!\}$. One way to remove the uncertainty is to construct a non-redundant polyhedron that gives (to ensure soundness) the same backward result by the concrete operator. Let $Y = \gamma_g(V_Y, R_Y)$ be a (possibly degenerate) polyhedron in generator form coming from one branch of the test. We can safely add to Y any ray $\vec{r}$ as long as it does not add any environment satisfying the guard, i.e., such that:

$$\tau\{\!|\, \vec{a}\cdot\vec{x} \geq b \,|\!\}\, \gamma_g(V_Y, R_Y \cup \{\vec{r}\}) = \tau\{\!|\, \vec{a}\cdot\vec{x} \geq b \,|\!\}\, \gamma_g(V_Y, R_Y) \tag{12}$$

as this implies $\overleftarrow{\tau}\{\!|\, \vec{a}\cdot\vec{x} \geq b \,|\!\}\, \gamma_g(V_Y, R_Y \cup \{\vec{r}\}) = \overleftarrow{\tau}\{\!|\, \vec{a}\cdot\vec{x} \geq b \,|\!\}\, \gamma_g(V_Y, R_Y)$. We choose to pick the rays $\vec{r}$ to add from those in the polyhedron Z from the other branch that satisfy (12). The effect is to create common rays in $\overleftarrow{\tau}^\sharp\{\!|\, \vec{a}\cdot\vec{x} \geq b \,|\!\}\, Y$ and $\overleftarrow{\tau}^\sharp\{\!|\, \vec{a}\cdot\vec{x} < b \,|\!\}\, Z$, and make the subsequent intersection as large as possible. Such degeneracies often occur in practice, in particular at loop exits. For instance, in Fig. 1, we must compute $\overleftarrow{\tau}\{\!|\, i < 100 \,|\!\}\, X_5 \cap \overleftarrow{\tau}\{\!|\, i \geq 100 \,|\!\}\, X_8$ where X_8 satisfies $i = 100$. In this case, the heuristics adds the ray $i - j$ to X_8. This is precise enough to yield the expected sufficient condition, i.e., $j \in [0;5]$ at (1). Note that, sometimes, Z does not contain any ray satisfying (12) (for instance when Z is empty) in which case the improvement heuristic cannot be used and we revert to the classic guard of Theorem 6. While useful, our heuristics is thus fragile and begs to be improved.

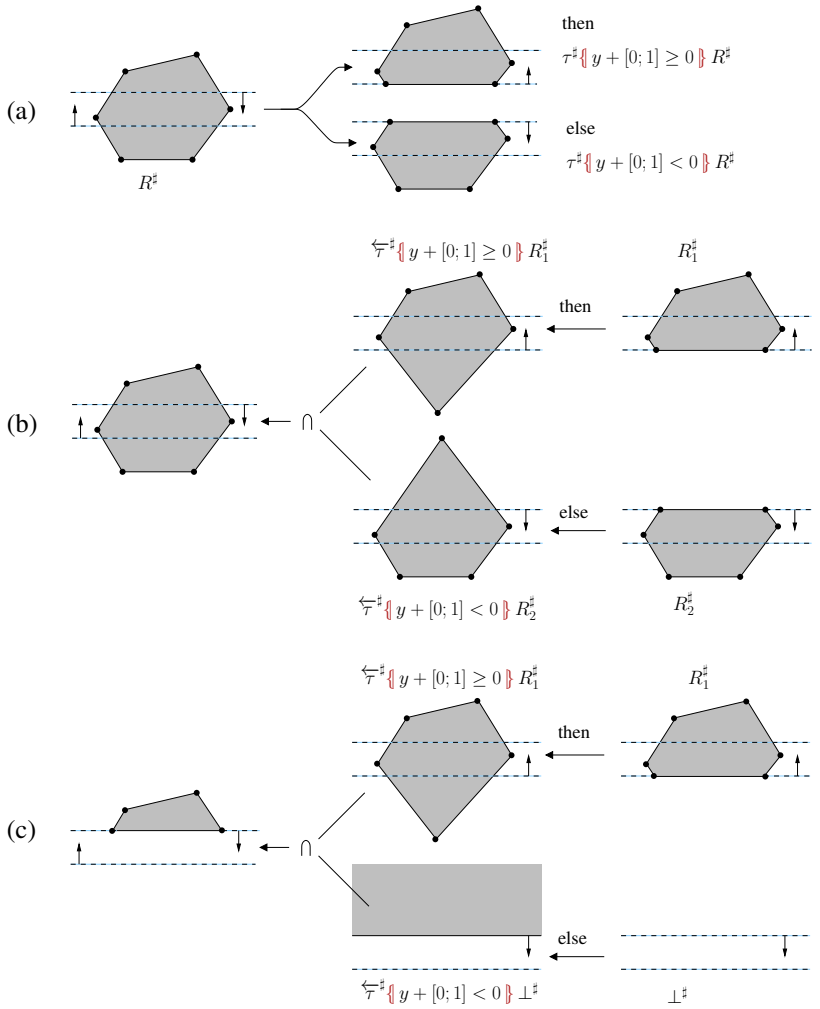

Figure 5: Forward (a) and backward (b)–(c) semantics of `if` $(y + [0;1] \geq 0)$ $\{ s_1 \}$ `else` $\{ s_2 \}$.

Alternate abstractions. In the preceding under-approximations of $\overleftarrow{\tau} \{\!| \vec{a} \cdot \vec{x} \geq b |\!\} R$, we chose to keep R intact in the result and add as many points from $\{ \rho \mid \vec{a} \cdot \vec{\rho} < b \}$ as possible. However, the abstract function $\overleftarrow{\tau}^\sharp \{\!| \vec{a} \cdot \vec{x} \geq b |\!\} R^\sharp$ is not monotonic in $R^\sharp$. In particular, reducing $R^\sharp$ may enable us to add more point from $\{ \rho \mid \vec{a} \cdot \vec{\rho} < b \}$, as shown in Fig. 6. There exists an infinite number of such under-approximations, and future work is required to design choice heuristics. There is, however, one interesting case: abstracting $\overleftarrow{\tau} \{\!| \vec{a} \cdot \vec{x} > b |\!\} R$ as $\{ \rho \mid \vec{a} \cdot \vec{\rho} \leq b \}$, ignoring R.[2] Figure 5.(c) shows the result of applying this abstraction in an `if` (b) $\{ s_1 \}$ `else` $\{ s_2 \}$ statement. Intuitively, this consists in choosing to ignore one branch, and thus returning only the states that necessarily enter one branch (this is also how `assert` (b) statements are effectively modeled).

3.3. Projection

Given a variable V, projecting V is a special form of assignment that forgets its value. Its concrete forward semantics is:

[2]Current polyhedral abstract domains also support strict constraints, which makes it possible to abstract $\overleftarrow{\tau} \{\!| \vec{a} \cdot \vec{x} \geq b |\!\} R$ similarly to $\{ \rho \mid \vec{a} \cdot \vec{\rho} < b \}$. Using $\{ \rho \mid \vec{a} \cdot \vec{\rho} \leq b \}$ in this case would not be sound.

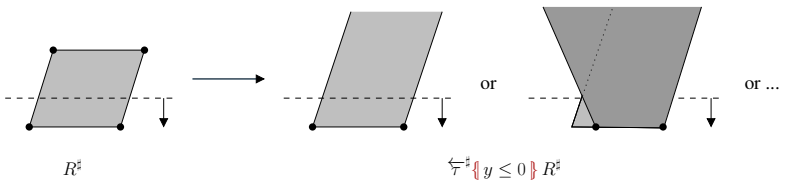

Figure 6: Alternate abstractions of guards.

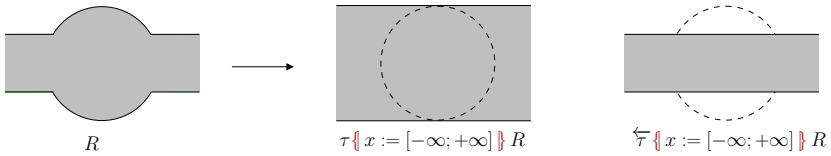

Figure 7: Forward and backward projection $x := [-\infty; +\infty]$.

$$\begin{aligned} \tau\{\!|\, V := [-\infty; +\infty] \,|\!\}\, R &\overset{\text{def}}{=} \{\, \rho[V \mapsto v] \mid \rho \in R, v \in \mathbb{R} \,\} \\ &= \{\, \rho \in \mathcal{E} \mid \exists v \in \mathbb{R} : \rho[V \mapsto v] \in R \,\} \end{aligned}$$

and so, the corresponding concrete backward semantics is:

$$\overleftarrow{\tau}\{\!|\, V := [-\infty; +\infty] \,|\!\}\, R = \{\, \rho \in \mathcal{E} \mid \forall v \in \mathbb{R} : \rho[V \mapsto v] \in R \,\} \ . \tag{13}$$

This operator is illustrated in Fig. 7. We have the following property:

Theorem 8.
If R is convex and closed, then $\overleftarrow{\tau}\{\!|\, V := [-\infty; +\infty] \,|\!\}\, R$ is either R or $\emptyset$.
Moreover, $\tau\{\!|\, V := [-\infty; +\infty] \,|\!\}\, R = R \iff \overleftarrow{\tau}\{\!|\, V := [-\infty; +\infty] \,|\!\}\, R = R$.

PROOF. See Appendix A.7. □

The backward projection can thus be efficiently and exactly implemented in any convex closed domain featuring an exact forward projection operator, which includes polyhedra, intervals, and octagons. We state:

$$\overleftarrow{\tau}^\sharp\{\!|\, V := [-\infty; +\infty] \,|\!\}\, X^\sharp \overset{\text{def}}{=} \begin{cases} X^\sharp & \text{if } \gamma(\tau^\sharp\{\!|\, V := [-\infty; +\infty] \,|\!\}\, X^\sharp) = \gamma(X^\sharp) \\ \bot^\sharp & \text{otherwise} \ . \end{cases} \tag{14}$$

The projection is also useful to model variable addition and removal. Although we assumed, for the sake of simplicity, that $\mathcal{V}$ is fixed, adding variables locally will come handy to model assignments. We thus define in the concrete forward semantics (assuming that added variables are not initialized and can take any value in $\mathbb{R}$):

$$\begin{aligned} \tau\{\!|\, \texttt{del}\ V \,|\!\}\, R &\overset{\text{def}}{=} \{\, \rho \mid \exists v \in \mathbb{R} : \rho[V \mapsto v] \in R \,\} \\ \tau\{\!|\, \texttt{add}\ V \,|\!\}\, R &\overset{\text{def}}{=} \{\, \rho[V \mapsto v] \mid \rho \in R,\ v \in \mathbb{R} \,\} \ . \end{aligned}$$

The following identities on the concrete backward semantics come easily:

$$\begin{aligned} \overleftarrow{\tau}\{\!|\, \texttt{del}\ V \,|\!\} &= \tau\{\!|\, \texttt{add}\ V \,|\!\} \\ \overleftarrow{\tau}\{\!|\, \texttt{add}\ V \,|\!\} &= \tau\{\!|\, \texttt{del}\ V \,|\!\} \circ \overleftarrow{\tau}\{\!|\, V := [-\infty; +\infty] \,|\!\} \end{aligned} \tag{15}$$

which allow modeling backward variable addition and removal in terms of their forward counterparts (which are exactly abstracted).

3.4. Assignments

Fallback operator. We first note that, in the forward semantics, the projection over-approximates any assignment: $\tau\{\!|\, V := \mathit{expr} \,|\!\} \subseteq \tau\{\!|\, V := [-\infty; +\infty] \,|\!\}$. By Theorem 3.6, we thus get $\overleftarrow{\tau}\{\!|\, V := [-\infty; +\infty] \,|\!\} \subseteq \overleftarrow{\tau}\{\!|\, V := \mathit{expr} \,|\!\}$: a sound backward projection can be used to under-approximate any assignment.

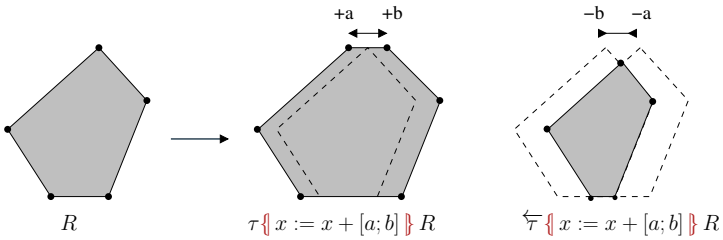

Figure 8: Forward and backward non-deterministic shift $x := x + [a; b]$.

Reduction to guard. More interestingly, general assignments can be reduced to guards by introducing a temporary variable V'. We denote by $[V/V']$ the renaming of V' as V (which is always capture-free as V should not occur in the argument). We have, in the concrete forward semantics:

$$\tau\{\!|\, V := e \,|\!\} = [V/V'] \circ \tau\{\!|\, \texttt{del}\ V \,|\!\} \circ \tau\{\!|\, V' = e \,|\!\} \circ \tau\{\!|\, \texttt{add}\ V' \,|\!\}$$

which implies in the concrete backward semantics, by (15), Theorem 3, and the fact that $\overleftarrow{[V/V']} = [V'/V]$:

$$\begin{array}{rcl} \overleftarrow{\tau}\{\!|\, V := e \,|\!\} & = & \overleftarrow{\tau}\{\!|\, \texttt{add}\ V' \,|\!\} \circ \overleftarrow{\tau}\{\!|\, V' = e \,|\!\} \circ \overleftarrow{\tau}\{\!|\, \texttt{del}\ V \,|\!\} \circ \overleftarrow{[V/V']} \\ & = & \tau\{\!|\, \texttt{del}\ V' \,|\!\} \circ \overleftarrow{\tau}\{\!|\, V' := [-\infty; +\infty] \,|\!\} \circ \overleftarrow{\tau}\{\!|\, V' = e \,|\!\} \circ \tau\{\!|\, \texttt{add}\ V \,|\!\} \circ [V'/V] \ . \end{array} \tag{16}$$

As the forward operators $\tau\{\!|\, \texttt{add}\ V \,|\!\}$ and $\tau\{\!|\, \texttt{del}\ V' \,|\!\}$, the projection, as well as $[V'/V]$, can be exactly modeled in our abstract domains, a sound under-approximation of $\overleftarrow{\tau}\{\!|\, V := e \,|\!\}$ can be defined using the under-approximations of guards presented earlier. Note that the temporary variable is only required if V occurs in e. Otherwise, $\tau\{\!|\, V := e \,|\!\}$ can be simplified as $\tau\{\!|\, V = e \,|\!\} \circ \tau\{\!|\, V := [-\infty; +\infty] \,|\!\}$, which yields directly:

$$\overleftarrow{\tau}\{\!|\, V := e \,|\!\} = \overleftarrow{\tau}\{\!|\, V := [-\infty; +\infty] \,|\!\} \circ \overleftarrow{\tau}\{\!|\, V = e \,|\!\} \ .$$

In case of a degenerate polyhedron in a guard argument, Sec. 3.2 relied on rays provided by another polyhedron to guide the operation. We do not have another polyhedron here, but we know that the guard is followed by a projection, hence, the heuristic is modified to use the rays $\vec{e}_{V'}$ and $-\vec{e}_{V'}$, where $\vec{e}_{V'}$ is the basis vector corresponding to the variable V'. Intuitively, we try to maximize the set of environments ρ such that the result of the guard contains $\{\, \rho[V' \mapsto v] \mid v \in \mathbb{R} \,\}$, and so, will be kept by $\overleftarrow{\tau}\{\!|\, V' := [-\infty; +\infty] \,|\!\}$.

Direct assignments. For some restricted, yet useful, classes of assignments, it is possible to express the backward semantics using only forward operators. This is the case, in particular, for purely non-deterministic assignments $V := [a; b]$ (generalizing the projection), for variable shifts $V := V + [a; b]$, and for variable copies $V := W$. We can use the following theorem:

Theorem 9.

1. *If R is a convex closed set, then:* $\overleftarrow{\tau}\{\!|\, V := V + [a; b] \,|\!\}\, R = (\tau\{\!|\, V := V - a \,|\!\}\, R) \cap (\tau\{\!|\, V := V - b \,|\!\}\, R)$.
2. *If R is a convex closed set, then:* $\overleftarrow{\tau}\{\!|\, V := [a; b] \,|\!\}\, R =$
   $(\tau\{\!|\, V := [-\infty; +\infty] \,|\!\} \circ (\tau\{\!|\, V := V - a \,|\!\} \cap \tau\{\!|\, V := V - b \,|\!\}) \circ \tau\{\!|\, V \geq a \wedge V \leq b \,|\!\})R$.
3. *If $V \neq W$, then:* $\overleftarrow{\tau}\{\!|\, V := W \,|\!\} = \tau\{\!|\, V := [-\infty; +\infty] \,|\!\} \circ \tau\{\!|\, V = W \,|\!\}$.

PROOF. See Appendix A.8. □

For instance, while the forward effect of a non-deterministic shift $V := V+[a; b]$ is to inflate its argument, its backward effect will shrink it, as illustrated in Fig. 8. In the polyhedra and the octagon domains, as well as the interval domain except for 3, all the forward operations involved in Theorem 9 are exact. As a consequence, Theorem 9 yields sound and exact backward operators.

Another interesting case is when the assigned expression e is invertible, i.e., there exists an expression, denoted e^{-1}, that allows recovering the initial value of the variable:

$$\tau\{\!|\, V := e^{-1} \,|\!\} \circ \tau\{\!|\, V := e \,|\!\} = \tau\{\!|\, V := e \,|\!\} \circ \tau\{\!|\, V := e^{-1} \,|\!\} = \lambda R.R \ .$$

Examples include: affine expressions $e = \sum_{W \in \mathcal{V}} \alpha_W W + \beta$ where V appears in the expression (i.e., $\alpha_V \neq 0$); then: $e^{-1} = (V - \beta - \sum_{W \in \mathcal{V} \setminus \{V\}} \alpha_W W)/\alpha_V$. We then have:

Theorem 10. *If $V := e$ is invertible then $\overleftarrow{\tau}\{\!|\, V := e \,|\!\} = \tau\{\!|\, V := e^{-1} \,|\!\}$.*

PROOF. This is a simple consequence of Theorem 3.8 stating that, if f and g are monotonic and $f \circ g = g \circ f = \lambda x.x$, then $\overleftarrow{f} = g$. □

When $\tau\{\!|\, V := e^{-1} \,|\!\}$ can be exactly modeled in the abstract domain, then it also gives a sound and exact abstraction of $\overleftarrow{\tau}\{\!|\, V := e \,|\!\}$; this is the case for instance if e is affine in the polyhedra domain. However, if $\tau^\sharp\{\!|\, V := e^{-1} \,|\!\}$ induces an over-approximation in the abstract domain (such as a general affine assignment in intervals or octagons), it cannot be used to soundly under-approximate $\overleftarrow{\tau}\{\!|\, V := e \,|\!\}$.

In practice, it is useful to decompose a complex assignment into a sequence of assignments that can be directly handled by Theorems. 9–10; for instance: $\overleftarrow{\tau}\{\!|\, V := e + [a;b] \,|\!\} = \overleftarrow{\tau}\{\!|\, V := e \,|\!\} \circ \overleftarrow{\tau}\{\!|\, V := V + [a;b] \,|\!\}$. When possible, this results in a better precision than using the general formula (16) as the general case uses a backward guard which often incurs some under-approximation.

3.5. Lower widenings

We now present lower widenings to compute fixpoints in finite time in the polyhedra, intervals, and octagons domains. Recall that a lower widening $\underline{\triangledown}$ must respect two properties (Definition 1): it must under-approximate $\cap$, and it must enforce termination. Lower widenings are introduced in [12] but, up to our knowledge, and unlike (upper) widenings, no practical instance on infinite domains has ever been designed. We stress on the fact that lower widenings are very different from narrowing operators $\triangle$ traditionally used in invariant inference. Both are used to accelerate the computation of decreasing iterations, but, while lower widenings under-approximate greatest fixpoints, narrowings refine an over-approximation of a least fixpoint. Lower widenings are designed to "jump below" fixpoints (in a similar way upper widenings "jump above" fixpoints), hence performing an induction, while narrowings "stay above" fixpoints (performing a refinement).

Polyhedra. The classic (upper) widening $\triangledown$ on polyhedra [15] consists in keeping stable constraints and removing unstable ones. By analogy, we propose a lower widening that keeps only stable generators. Let V_P and R_P denote the vertices and rays of a polyhedron $P = \gamma_g(V_P, R_P)$. We define $\underline{\triangledown}$ formally as:

$$\begin{array}{l} V_{A \underline{\triangledown} B} \stackrel{\text{def}}{=} \{\, \vec{v} \in V_A \mid \vec{v} \in B \,\} \\ R_{A \underline{\triangledown} B} \stackrel{\text{def}}{=} \{\, \vec{r} \in R_A \mid B \oplus \mathbb{R}^+ \vec{r} = B \,\} \end{array} \tag{17}$$

where checking the stability of rays involves the Minkowski sum operator $\oplus$, defined as $A \oplus B \stackrel{\text{def}}{=} \{\, \vec{a} + \vec{b} \mid \vec{a} \in A, \vec{b} \in B \,\}$, and $\mathbb{R}^+ \vec{r} \stackrel{\text{def}}{=} \{\, \lambda \vec{r} \mid \lambda \geq 0 \,\}$. We have:

Theorem 11. *$\underline{\triangledown}$ is a lower widening.*

PROOF. See Appendix A.9. □

This lower widening is illustrated in Fig. 9.

Generator representations are not unique, and the output of $\underline{\triangledown}$ depends on the choice of representation. The same issue occurs for the standard widening. We can use a similar fix: we add to $A \underline{\triangledown} B$ any generator from B that is redundant with a generator in A.

The lower widening can also be refined in a classic way by permitting thresholds: given a finite set of vertices (resp. rays), each vertex $\vec{v}$ (resp. ray $\vec{r}$) included in both polyhedra A and B ($\vec{v} \in A \wedge \vec{v} \in B$, resp. $A \oplus \mathbb{R}^+ \vec{r} = A \wedge B \oplus \mathbb{R}^+ \vec{r} = B$) is added to $A \underline{\triangledown} B$.

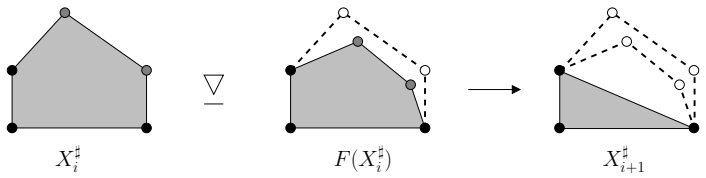

Figure 9: Lower widening on polyhedra: unstable vertices (in gray) are removed.

Intervals. The classic (upper) widening on intervals [14] simply sets unstable bounds to $+\infty$ or $-\infty$. This is generally too coarse and, in practice, the interval widening also uses thresholds [13]: a finite set of predefined bounds is tested for stability before bailing out to infinity. For our lower widening, we also assume a finite set T of thresholds. The widening will then increase unstable lower bounds and decrease unstable upper bounds up to the next threshold, if possible:

$$\begin{array}{l} [a;b] \mathbin{\underline{\triangledown}} [c;d] \quad \stackrel{\text{def}}{=} \quad \text{let } L \stackrel{\text{def}}{=} \{\, t \in T \cup \{a\} \mid a, c \leq t \,\} \text{ in} \\ \qquad\qquad\qquad\quad \text{let } H \stackrel{\text{def}}{=} \{\, t \in T \cup \{b\} \mid t \leq b, d \,\} \text{ in} \\ \qquad\qquad\qquad\quad \begin{cases} [\min L; \max H] & \text{if } L, H \neq \emptyset,\ \min L \leq \max H \\ [c;c] & \text{otherwise if } a \leq c = d \leq b \\ [b;b] & \text{otherwise if } a < c \leq d = b \\ [a;a] & \text{otherwise if } a = c \leq d < b \\ \emptyset & \text{otherwise} \end{cases} \end{array} \tag{18}$$

When this is not possible (i.e., the intervals do not contain a threshold), we first try to construct a singleton before bailout out to $\emptyset$. We naturally choose $[c;d]$ if it is a singleton. Otherwise, if a single bound is stable, we use it as singleton.

Theorem 12. *$\underline{\triangledown}$ is a lower widening.*

PROOF. See Appendix A.10. □

Octagons. The classic way [16] to construct an (upper) widening on octagons consists in considering an octagon as a conjunction of constraints of the form $\pm X \pm Y \in [a;b]$ and applying an interval widening independently on each constraint. The exact same method can be used to construct a lower widening.

As upper widenings, lower widenings perform dynamic approximations to solve an undecidable problem: although they work on infinitely many programs [35], they also fail on infinitely many ones. Devising new widenings adapted to specific program classes is an important part of abstract domain design (see for instance [36, 37] for advanced upper widenings on polyhedra and octagons). Our goal here was only to prove that lower widenings do exist by constructing a few simple ones. We leave the design of more complex lower widenings for future work.

3.6. Expression approximation

We saw in Sections 3.2 and 3.4 that affine expressions play a special role when trying to abstract guards and assignments: they are easier to abstract precisely because polyhedra can represent affine constraints exactly. When encountering non-affine numeric expressions, one solution is to use one of the coarse fallback operators we proposed. When over-approximating forward operators, we proposed in [38] a more precise solution consisting in abstracting non-affine expressions into affine ones with some non-determinism embedded in a constant interval (or constant coefficients). We now show that the very same principle can be applied to under-approximate backward operators.

Formally, we denote by $e \sqsubseteq_D f$ the fact that a numeric or boolean expression f approximates e on an environment set $D \subseteq \mathcal{E}$, i.e.:

$$e \sqsubseteq_D f \quad \stackrel{\text{def}}{\iff} \quad \forall \rho \in D : [\![e]\!]\rho \subseteq [\![f]\!]\rho \ .$$

We showed in [38] that, if $e \sqsubseteq_D f$ and $R \subseteq D$, then:

$$\begin{array}{lll} \tau\{\!|\, V := e \,|\!\}\, R & \subseteq & \tau\{\!|\, V := f \,|\!\}\, R \\ \tau\{\!|\, e \,|\!\}\, R & \subseteq & \tau\{\!|\, f \,|\!\}\, R \end{array}$$

i.e., e can be replaced with f in forward assignments and tests, which results in a sound over-approximation. For backward operators, we introduce the following properties:

Theorem 13. *If $e \sqsubseteq_D f$, then, for any $R \subseteq \mathcal{E}$:*

1. $\overleftarrow{\tau}\{\!|\, V := e \,|\!\}\, R \supseteq (\overleftarrow{\tau}\{\!|\, V := f \,|\!\}\, R) \cap D$.
2. $\overleftarrow{\tau}\{\!|\, e \,|\!\}\, R \supseteq (\overleftarrow{\tau}\{\!|\, f \,|\!\}\, R) \cap D$.

PROOF. See Appendix A.11.

As a consequence, we can under-approximate $\overleftarrow{\tau}\{\!|\, V := e \,|\!\}\, \gamma(R^\sharp)$ in an abstract domain as $(\overleftarrow{\tau}^\sharp\{\!|\, V := f \,|\!\}\, R^\sharp) \cap^\sharp D^\sharp$ instead of $\overleftarrow{\tau}^\sharp\{\!|\, V := e \,|\!\}\, R^\sharp$ whenever $e \sqsubseteq_{\gamma(D^\sharp)} f$. Similarly, $\overleftarrow{\tau}\{\!|\, e \,|\!\}\, \gamma(R^\sharp)$ will be abstracted as $(\overleftarrow{\tau}^\sharp\{\!|\, f \,|\!\}\, R^\sharp) \cap^\sharp D^\sharp$. It remains to discuss the choice of f and $D^\sharp$. Our goal is to make the transformed statement easier to handle in our abstract domain. Hence, we exploit the so-called "linearization" technique [38], which converts an arbitrary expression into an expression of the form $\sum_{V \in \mathcal{V}} \alpha_V V + [a; b]$ by performing interval arithmetic on non-linear expression parts. The transformation is parametrized by a set of variable bounds, which can be extracted from $D^\sharp$, and outputs an expression f satisfying $e \sqsubseteq_{\gamma(D^\sharp)} f$. For instance, it will transform $\overleftarrow{\tau}^\sharp\{\!|\, X := Y \times Z \,|\!\}\, R^\sharp$ into $(\overleftarrow{\tau}^\sharp\{\!|\, X := 0.5 \times Y + [-0.5; 0.5] \,|\!\}\, R^\sharp) \cap^\sharp D^\sharp$ if $\gamma(D^\sharp)$ implies that $Y, Z \in [0; 1]$. On the one hand, a smaller $D^\sharp$ ensures that the expression f is tighter, and so, $\overleftarrow{\tau}\{\!|\, V := f \,|\!\}$ is more precise, i.e., provides a larger under-approximation. On the other hand, a smaller $D^\sharp$ mechanically reduces the precision of $(\overleftarrow{\tau}^\sharp\{\!|\, V := f \,|\!\}\, R^\sharp) \cap^\sharp D^\sharp$ due to the intersection with $D^\sharp$. In order to maximize the result of the backward assignment, the intersection should avoid discarding states from $\overleftarrow{\tau}\{\!|\, V := f \,|\!\}\, \gamma(R^\sharp)$. A practical solution is to use as $D^\sharp$ the result of a prior invariance analysis. Indeed, we know that, in the concrete, $\overleftarrow{\tau}\{\!|\, V := e \,|\!\}\, \gamma(R^\sharp) \subseteq \gamma(D^\sharp)$ as the concrete backward sufficient condition is always a subset of the concrete reachable states.

It may seem counter-intuitive that *over-approximating* expressions results in *under-approximating* backward transfer functions. Observe that over-approximations enlarge the non-determinism of expressions, and so, make it less likely to find sufficient conditions holding for all cases.

3.7. Integers

Up to now, we have assumed a real semantics, with environments in $\mathcal{V} \to \mathbb{R}$. We now consider an integer version of the semantics, denoted as $\overleftarrow{\tau}_{\mathbb{Z}}\{\!|\, s \,|\!\}$. Only three modifications are required compared to Fig. 3: environments live in $\mathcal{E} \stackrel{\text{def}}{=} \mathcal{V} \to \mathbb{Z}$, constants are integer $[\![\, [a; b] \,]\!]_{\mathbb{Z}}\, \rho \stackrel{\text{def}}{=} \{\, x \in \mathbb{Z} \mid a \leq x \leq b \,\}$, and the result of v_1/v_2 is truncated in $[\![\, e_1/e_2 \,]\!]_{\mathbb{Z}}\, \rho$. For the sake of simplicity, we assume division-free expressions in the rest of this section (in practice, an integer division a/b can be abstracted using a real-division and modeling truncation as a shift by some value in a non-deterministic interval: $a/b + [-1; 1]$; this follows the expression approximation scheme of Sec. 3.6).

Forward abstraction. When it comes to constraints-based abstractions (such as polyhedra, integers or octagons), a set C of constraints now represents only the integer points $\gamma_{\mathbb{Z}}(C)$ that satisfy them:

$$\gamma_{\mathbb{Z}}(C) \stackrel{\text{def}}{=} \gamma_c(C) \cap \mathbb{Z}^n \quad .$$

Forward abstract transfer functions can be constructed using the following observation: *Any sound forward abstraction for the real semantics is also a sound forward abstraction for the integer semantics.* However, exactness is not always maintained. For instance, projection on polyhedra is exact with respect to γ_c, but not with respect to $\gamma_{\mathbb{Z}}$ (as projecting integer sets gives rise to congruence relations that cannot be represented in polyhedra). In practice [39], operator applications are followed by a refinement phase that exploits integerness properties to tighten constraints (i.e., applies transformations that are sound with respect to $\gamma_{\mathbb{Z}}$ but not γ_c).

Backward abstraction. We now discuss this tactic for backward under-approximating operators. For affine guards $\vec{a} \cdot \vec{x} \geq b$, both (11) and Theorem 6 are still valid when considering the integer semantics:

Theorem 14. $\gamma_{\mathbb{Z}}(C \setminus \{\vec{a} \cdot \vec{x} \geq b\}) \subseteq \overleftarrow{\tau}_{\mathbb{Z}}\{\!|\, \vec{a} \cdot \vec{x} \geq b \,|\!\}\, \gamma_{\mathbb{Z}}(C)$.

PROOF. The proof is identical to that of Theorem 6. □

Hence, the operators from Sec. 3.2 can be soundly reused as-is. Some improvement in precision can be achieved by abstracting $\overleftarrow{\tau_{\mathbb{Z}}}\{\!|\, \vec{a}\cdot\vec{x} > b \,|\!\}$ as $\overleftarrow{\tau_{\mathbb{Z}}}\{\!|\, \vec{a}\cdot\vec{x} \geq b+1 \,|\!\}$ instead of $\overleftarrow{\tau_{\mathbb{Z}}}\{\!|\, \vec{a}\cdot\vec{x} \geq b \,|\!\}$ (this optimization was used implicitly in the example of Fig. 1). Moreover, instead of removing only the constraints that are redundant with $\vec{a}\cdot\vec{x} \geq b$ with respect to γ_c, it becomes possible to remove redundant constraints with respect to $\gamma_{\mathbb{Z}}$. This removes a larger number of constraints, yielding a larger (i.e., more precise) under-approximation, but it requires the use of integer linear programming methods, which have a high complexity. In practice, a cost versus precision trade-off would need to be reached, which we leave as future work.

The backward integer projection is:

$$\overleftarrow{\tau_{\mathbb{Z}}}\{\!|\, V := [-\infty;+\infty] \,|\!\}\, R = \{\, \rho \in \mathcal{V}\to\mathbb{Z} \mid \forall v\in\mathbb{Z} : \rho[V\mapsto v]\in R \,\}\ .$$

We have the following slight variant of Theorem 8:

Theorem 15. *If R is convex closed in $\mathbb{R}^n$, then:*

$$\overleftarrow{\tau_{\mathbb{Z}}}\{\!|\, V := [-\infty;+\infty] \,|\!\}(R\cap\mathbb{Z}^n) = \begin{cases} R\cap\mathbb{Z}^n & \text{if } \tau\{\!|\, V := [-\infty;+\infty] \,|\!\}\, R = R \\ \emptyset & \text{otherwise}\ . \end{cases}$$

PROOF. The proof stems from the fact that, if R is convex, $\forall v\in\mathbb{Z} : \rho[V\mapsto v]\in R \iff \forall v\in\mathbb{R} : \rho[V\mapsto v]\in R$.

Hence, we can reuse as-is the backward projection operator (14). This operator still performs a check using the exact forward projection in the real field, $\gamma(\tau^\sharp\{\!|\, V := [-\infty;+\infty] \,|\!\}\, X^\sharp) = \gamma(X^\sharp)$, but actually computes an exact backward projection for integers.

Finally, the general formula for backward assignments (16) is still valid in the context of an integer semantics. The special assignments described in Theorems 9 and 10 are also valid when replacing $\tau\{\!|\, V := e \,|\!\}$ and $\overleftarrow{\tau}\{\!|\, V := e \,|\!\}$ with $\tau_{\mathbb{Z}}\{\!|\, V := e \,|\!\}$ and $\overleftarrow{\tau_{\mathbb{Z}}}\{\!|\, V := e \,|\!\}$. Recall that these cases reduce the backward semantics to a forward one; hence, after abstraction, soundness of the backward function requires the exactness of the forward operators used (as the over-approximation must also be an under-approximation). We should then be careful that an exact forward abstract operator for γ_c may not be exact for $\gamma_{\mathbb{Z}}$ (one example is the projection operator).

We have demonstrated in this section that, the same way an abstract domain for $\mathbb{R}^n$ can be used to abstract an integer forward semantics, the backward operators we proposed for $\mathbb{R}^n$ can also be used to find sufficient conditions on integer programs. The same caveat applies: while soundness is easy to achieve, ensuring more precision requires taking into account the integerness properties.

3.8. Joins and disjunctions

Joins. In invariance analyses, unions of environment sets are computed at every control-flow join. Naturally, a large effort in abstract analysis design is spent designing precise and efficient over-approximating abstractions of unions. By the duality of Theorem 3.2, such joins do not occur in sufficient condition analyses; they are replaced with intersections $\cap$ at control-flow splits, which are extremely easy to abstract in many domains (such as polyhedra, octagons, and intervals). Hence, we avoid the thorny issue of designing under-approximations of *arbitrary* unions. We do under-approximate unions as part of guard operators (Sec. 3.2), but these have a very specific form which helped us design sensible approximations.

Disjunctive completion. Convex domains, such as polyhedra, cannot exactly represent unions, which must be over-approximated (e.g., by the convex hull for polyhedra). When the incurred loss of precision is unacceptable, one solution is to use a disjunctive completion [40], i.e., maintain a set of polyhedra $X_1^\sharp,\ldots,X_m^\sharp$ that represent symbolically the union of several polyhedra:

$$\gamma_d(X_1^\sharp,\ldots,X_m^\sharp) \stackrel{\text{def}}{=} \bigcup_i \gamma(X_i^\sharp)\ .$$

This idea can be carried to sufficient condition analysis. Together with the support for strict constraints, symbolic disjunctions allow representing exactly the semantics of guards $\overleftarrow{\tau}\{\!|\, \vec{a}\cdot\vec{x}\geq b \,|\!\}\, R = R\cup\{\,\rho\in\mathcal{E} \mid \vec{a}\cdot\vec{\rho} < b\,\}$; we simply add a new polyhedron representing $\vec{a}\cdot\vec{\rho} < b$ to the abstraction of R (we assume here that our polyhedron

domain can express exactly strict constraints, which is the case of most modern ones [39]). In an invariance analysis using disjunctive completions, forward operators are applied independently to each element, which is justified by the fact that concrete forward operators are $\cup$-morphisms. We can also apply backward operators element-wise; the soundness comes from the sup-$\cup$-morphism property of the concrete semantics (Theorem 2.2):

$$\overleftarrow{\tau}\{\!|\, s \,|\!\}(\bigcup_i X_i) \supseteq \bigcup_i \overleftarrow{\tau}\{\!|\, s \,|\!\}\, X_i \tag{19}$$

hence, we have:

$$\begin{array}{ll} & (\overleftarrow{\tau}\{\!|\, s \,|\!\} \circ \gamma_d)(X_1^\sharp, \ldots, X_m^\sharp) \\ = & \overleftarrow{\tau}\{\!|\, s \,|\!\}(\bigcup_i \gamma(X_i^\sharp)) \\ \supseteq & \bigcup_i (\overleftarrow{\tau}\{\!|\, s \,|\!\} \circ \gamma) X_i^\sharp \\ \supseteq & \bigcup_i (\gamma \circ \overleftarrow{\tau}^\sharp\{\!|\, s \,|\!\}) X_i^\sharp \\ = & (\gamma_d \circ \overleftarrow{\tau}^\sharp\{\!|\, s \,|\!\})(X_1^\sharp, \ldots, X_m^\sharp) \ . \end{array}$$

Note that (19) is generally not an equality: unlike invariant analyses, partitioning the environment set already induces a loss of precision in the concrete. The main practical concern when using a disjunctive completion remains to limit the number of abstract elements. A classic technique in over-approximating invariant analyses consists in replacing some abstract elements with their abstract join when their number exceeds a threshold. We can use existing heuristics to choose which disjoints to keep and which to abstract but, as we are interested in under-approximations, we would discard abstract elements instead of joining them. Future work is required to tune such strategies to the need of sufficient condition analyses.

We stress on the fact that, unlike other analyses (such as [19]) that fully rely on disjunctive completions, our use of disjunctive completion is optional: it can be used to gain precision, but it is not necessary to achieve soundness.

4. Semantic extensions

In the two preceding sections, we stated and abstracted the problem of inferring sufficient conditions for a program on its initial states so that all its executions obey a given global program invariant. In this section, we broaden a little the scope of the problem by discussing the inference of inevitability conditions and of conditions on the program environment. More precisely, we show that these can be reduced, in certain limited cases, to the problem solved previously.

4.1. Inferring inevitability

Recall our original statement of the sufficient condition inference problem on transition systems (Sec. 2.1): given an initial set $I \subseteq \Sigma$ and the error state $\omega \in \Sigma$, we compute $I \cap \mathrm{cond}(\Sigma \setminus \{\omega\})$, which corresponds to the initial states in I that guarantee that no computation reaches the error state ω.

Instead of requiring that a system always satisfies an invariant, we may require that it enters a certain state after some time. We can transform this problem into an invariance one by adding a flag variable f set non-deterministically and asserting the condition. For instance, given the program $\texttt{while}\ (b)\ \{\, P \,\}$ and a condition c, we would construct:

$$\texttt{while}\ (b)\ \{\, \texttt{if}\ ([0;1])\ \{\, f := 1 \,\}; \texttt{assert}\ (c \vee \neg f); P \,\}; \texttt{assert}\ (f = 1)$$

Then, every execution starting in $\mathrm{cond}(\Sigma \setminus \{\omega\})$ never encounters any error and, either reaches a state satisfying c, or never reaches the end of the program.

This construction is not sufficient to prove inevitability properties: we need to exclude initial states with executions that do not reach the end of the program (i.e., they loop forever on non-error states). We use an idea by Cousot and Cousot [28] and enrich the transition system with a counter variable l counting execution steps down starting from some positive value and stopping at 0 when reaching the end of the program. More precisely, given the transition system (Σ, τ) and target set $T \subseteq \Sigma$, we construct (Σ', τ') and T' as follows:

$$\begin{array}{l} \Sigma' \stackrel{\text{def}}{=} \Sigma \times \mathbb{N} \\ ((\sigma, l), (\sigma', l')) \in \tau' \iff^{\text{def}} ((\sigma \notin T \wedge (\sigma, \sigma') \in \tau) \vee \sigma = \sigma' \in T) \wedge l = l' + 1 \\ T' \stackrel{\text{def}}{=} \{\, (\sigma, l) \in \Sigma' \mid l > 0 \vee \sigma \in T \,\} \end{array}$$

and compute $\mathrm{cond}(T')$ in (Σ', τ'). The following theorem is similar to Theorem 1, but targets inevitability instead of invariance:

Theorem 16.
If $(\sigma, l) \in \mathrm{cond}(T')$, then all the traces starting in σ eventually enter a state in T.
If the non-determinism in τ is finite, the converse holds.

PROOF. See Appendix A.12.

Due to the requirement on the non-determinism in τ, the transformation is not always complete. Completeness requires a finite non-determinism, i.e. $\forall \sigma : \mathrm{post}(\{\sigma\})$ is finite (note that this is weaker than requiring a bounded non-determinism, i.e., $\exists n \in \mathbb{N} : \forall \sigma : |\mathrm{post}(\{\sigma\})| \leq n$). In particular, the finiteness restriction prevents completeness when analyzing fair systems, as an infinite number of countable choices must be performed, e.g.:

$$\texttt{while}\ ([0;1])\ \{\, P; n := [-\infty;\infty]; \texttt{while}\ (n > 0)\ \{\, Q; n := n - 1 \,\}\ \}\ .$$

However, if the number of infinite choices is bounded, they can be embedded as fresh non-initialized variables to obtain a program with finite non-determinism. It remains to develop abstract domains able to reason precisely on the program counter l, which we leave as future work (we can look at [28, 41] for inspiration).

4.2. Inferring conditions on the environment

In our semantics, for a program to be correct, it must never fail whatever the sequence of values chosen at each non-deterministic choice. Intuitively, the semantics has no control on the sequence of non-deterministic values and must thus assume the worst. We can additionally support another kind of non-determinism, called inputs, that the semantics can control to steer the program and prevents its failure. We wish to infer sufficient conditions on the inputs such that any sequence of inputs satisfying the conditions leads to an error-free execution. For instance, given the integer program:

$$\texttt{while}\ (x > 0)\ \{\, i := \texttt{input}(); x := x - i \,\}$$

that decrements x by a user-specified amount at each step until it reaches 0, we wish to infer sufficient conditions on the sequence of inputs (in addition to the initial state) so that $x \geq 0$, is always satisfied. We suggest transforming this problem into an inference problem on initial states only, by adding extra variables, a and b, representing the bounds of the input statements, and inferring sufficient conditions on these bounds. The program becomes:

$$\texttt{while}\ (x > 0)\ \{\, i := [-\infty;\infty]; \texttt{assume}\ (i \geq a \wedge i \leq b); x := x - i \,\}$$

and the semantics $\mathrm{cond}(\{\, \rho \mid \rho(x) \geq 0 \,\})$ infers that $x = 0 \vee (x > 0 \wedge b \leq 1)$ is a sufficient initial condition. This transformation is not complete: we only look for uniform sufficient conditions (i.e., conditions that do not depend on the loop iteration). More work is required to infer non-uniform, or temporal conditions.

Game semantics. We can view the sufficient condition inference problem as a two player game: the Player controls initial states and the value of inputs, and forces the program to satisfy an invariant, while the Opponent controls non-deterministic choices, and seeks to invalidate the invariant. The analogy stops here because our framework is currently limited to making the Player choose all its moves in advance, negating completely the interleaving of player moves that makes the gist of game semantics.

5. Implementation

We have implemented the abstract analysis described in Sec. 3, which infers sufficient conditions on the initial states of a program so that it never violates any user-specified assertion. Our implementation is currently a simple proof-of-concept: it analyzes a toy language, featuring only numeric integer and real expressions. Moreover, it does not support the semantics extensions described in Sec. 4.

	invariant	backward iteration 1 ($\cap^\sharp$)	backward iteration 2 ($\underline{\triangledown}$)
(9)	$\mathtt{i} = 100 \wedge \mathtt{j} \in [0; 105]$	$\mathtt{i} = 100 \wedge \mathtt{j} \in [0; 105]$	$\mathtt{i} = 100 \wedge \mathtt{j} \in [0; 105]$
(8)	$\mathtt{i} = 100 \wedge \mathtt{j} \in [0; 110]$	$\mathtt{i} = 100 \wedge \mathtt{j} \in [0; 105]$	$\mathtt{i} = 100 \wedge \mathtt{j} \in [0; 105]$
(7)	$\mathtt{i} \in [1; 100] \wedge \mathtt{j} \geq 0 \wedge \mathtt{i} - \mathtt{j} \geq -10$	$\mathtt{i} \in [1; 100] \wedge \mathtt{j} \geq 0 \wedge \mathtt{i} - \mathtt{j} \geq -10$	$\mathtt{i} \in [1; 100] \wedge \mathtt{j} \geq 0 \wedge \mathtt{i} - \mathtt{j} \geq -5$
(6)	$\mathtt{i} \in [1; 100] \wedge \mathtt{j} \geq 0 \wedge \mathtt{i} - \mathtt{j} \geq -9$	$\mathtt{i} \in [1; 100] \wedge \mathtt{j} \geq 0 \wedge \mathtt{i} - \mathtt{j} \geq -9$	$\mathtt{i} \in [1; 100] \wedge \mathtt{j} \geq 0 \wedge \mathtt{i} - \mathtt{j} \geq -4$
(5)	$\mathtt{i} \in [0; 99] \wedge \mathtt{j} \geq 0 \wedge \mathtt{i} - \mathtt{j} \geq -10$	$\mathtt{i} \in [0; 99] \wedge \mathtt{j} \geq 0 \wedge \mathtt{i} - \mathtt{j} \geq -10$	$\mathtt{i} \in [0; 99] \wedge \mathtt{j} \geq 0 \wedge \mathtt{i} - \mathtt{j} \geq -5$
(4)	$\mathtt{i} \in [0; 100] \wedge \mathtt{j} \geq 0 \wedge \mathtt{i} - \mathtt{j} \geq -10$	$\mathtt{i} \in [0; 100] \wedge \mathtt{j} \geq 0 \wedge \mathtt{i} - \mathtt{j} \geq -5$	$\mathtt{i} \in [0; 100] \wedge \mathtt{j} \geq 0 \wedge \mathtt{i} - \mathtt{j} \geq -5$
(3)	$\mathtt{i} = 0 \wedge \mathtt{j} \in [0; 10]$	$\mathtt{i} = 0 \wedge \mathtt{j} \in [0; 10]$	$\mathtt{i} = 0 \wedge \mathtt{j} \in [0; 5]$

Figure 10: Analysis of the program from Fig. 1 with our prototype, showing the result of the invariance analysis and the subsequent two iterations of backward analysis.

Analyzer. The analyzer is based on the polyhedra abstract domain and computes a big-step semantics similar to (10) by induction on the program syntax. The program is analyzed in two phases. Firstly, the analyzer performs a standard over-approximating forward analysis using the classic polyhedra domain, and stores polyhedral invariants at each program point. Secondly, it performs an under-approximating backward analysis using the polyhedra operators described in Sec. 3. The invariants gathered during the first pass are used at two key locations. Firstly, to perform abstractions of non-linear expressions, as described in Sec. 3.6. Secondly, to bootstrap greatest fixpoint computations, using a meaningful over-approximation instead of $\top^\sharp$. We exploit here the fact that the sufficient conditions we compute are always subsets of the reachable states. To analyze loops, we employ our simple lower widening (17) that removes unstable generators (without threshold). We also employ classic techniques to improve the analysis of loops, which translate easily to sufficient condition analysis: unrolling the first few loop iterations, replacing the first applications of $\triangledown$ (resp. $\underline{\triangledown}$) with $\cup^\sharp$ (resp. $\cap^\sharp$), and refining the fixpoint abstraction with a few extra loop iterations without (upper nor lower) widening.

All the abstract operators are implemented using the Apron library [39]. Although Apron only supports over-approximating operators for forward and backward invariant analyses, it did not prove difficult to synthesize the under-approximating sufficient condition operators as a combination of exact polyhedra operators (including conversions between constraints and generators and redundancy removal) and some direct manipulation of the constraint and generator representations.

The analyzer is freely available in source form and also features a web-based interface [42].

Analysis examples. Our prototype is able to analyze our motivating example from Fig. 1. The polyhedra computed at each program point during the analysis are presented in Fig. 10. The first column presents the result of the invariance analysis (we do not detail the forward iterates with upper widening, which are standard). Then, the analysis performs two iterations of backward analysis, shown in the second and third column, before stabilizing. Note that the first iteration uses a meet $\cap^\sharp$ instead of the lower widening $\underline{\triangledown}$. The most salient point is the derivation of $\mathtt{i} - \mathtt{j} \geq -5$ at line (7) in the last column, which corresponds to the analysis of the loop guard `i < 100`: $\overleftarrow{\tau}^\sharp \{\!|\, \mathtt{i} \leq 99 \,|\!\} X_5^\sharp \cap^\sharp \overleftarrow{\tau}^\sharp \{\!|\, \mathtt{i} \geq 100 \,|\!\} X_8^\sharp$. This computation incorporates information from the assertion following the loop into the loop invariant. It is then only a matter of setting `i` to its value upon entering the loop, 0, to get the property we seek: $\mathtt{j} \leq 5$.

As second example we consider the Bubble Sort program in Fig. 11, which already illustrated the seminal work of Cousot and Halbwachs [15] on invariant inference using polyhedra. The sorting algorithm is abstracted by removing all array accesses; they are replaced with bound check assertions, while array element comparisons are replaced with a non-deterministic choice.

Analyzing this program gives, as invariant at the program end (20): $\mathtt{B} \leq \mathtt{N}$, and, as sufficient condition at the program entry (1): $\top^\sharp$. We have thus proved that the program never performs any array bound error, which is to be expected.

To motivate the use of sufficient conditions for program verification, we analyze next an incorrect version of the program, were the `B` is initialized to `N+1` instead of `N` at (1). Our analysis then finds that $\mathtt{N} \leq 0$ is a sufficient condition for the program to be correct. Indeed, a sure way to ensure the absence of error is to prevent the inner loop (containing the assertion) to be executed, which is ensured by this condition. In fact, this is the optimal condition as, when $\mathtt{N} \geq 1$, the program necessarily enters the inner loop and performs a failed bound check. Note that the inferred condition

```
(1) B = N;
(2) while (3) (B >= 1) {
    (4) J = 1;
    (5) T = 0;
    (6) while (7) (J <= B - 1) {
        (8) assert (J >= 1 && J <= N && J + 1 >= 1 && J + 1 <= N);
        (9) if ([0;1] == 1) { (10) T = J (11) } else { (12) };
        (13) J = J + 1
    (14) }
    (15) if (T == 0) { (16) return (17) } else { (18) };
    (19) B = T
(20) }
```

Figure 11: Bubble Sort example form [15] and analyzed by our prototype.

gathers program executions that never enter the outer loop (when $\mathtt{N} < 0$) and executions that perform an iteration of the outer loop (when $\mathtt{N} = 0$): it has merged information coming from distinct program paths. Finally, we analyze the same incorrect program but negate the assertion in order to force an array bound check error. In that case, our analysis infers that $\mathtt{N} = 1$ is a sufficient condition so that, whenever (8) is executed, an array bound check failure occurs. In fact, this indeed corresponds to a counter-example (although our analysis does not currently infer that (8) is reached). Moreover, note that, in fact, any value $\mathtt{N} \geq 1$ is bound to result in a bound check error. Even though the precise, concrete sufficient condition $\mathtt{N} \geq 1$ can be expressed in the polyhedra abstract domain, the abstract analysis returns a strict under-approximation: $\mathtt{N} = 1$. This loss of precision is caused by our use of non-exact abstract operators during the analysis.

6. Comparison with related work

Abstract interpretation [12] is concerned with all forms of exact and approximate abstract reasoning on systems and programs. However, the vast majority of applications are concerned with over-approximations, including the design of many numeric abstract domains able to infer a variety of invariants with different cost versus precision trade-offs. There are fewer works on sufficient condition inference, especially using under-approximations. In his work on abstract debugging, Bourdoncle [17] introduces sufficient conditions, denoted *always*(T), but only focuses on deterministic systems (i.e., $\widetilde{\mathrm{pre}} = \mathrm{pre}$). He mentions that classic domains, such as intervals, are inadequate to express under-approximations as they are not closed under complementation, but he does not propose an alternative. Moy [19] solves this issue by allowing disjunctions of abstract states, which corresponds to path enumerations and can grow arbitrarily large. Lev-Ami et al. [21] derive under-approximations from over-approximations by assuming that abstract domains are closed by complementation (or negation, when seen as formulas). Brauer et al. [3] employ boolean formulas on a bit-vector domain, which is necessarily finite, and employ a SAT solver to generate under-approximations by quantifier elimination. These approaches rely on very expressive (and so, costly) abstractions, and cannot be applied to more scalable ones, such as intervals or octagons.

Our approach consists instead in choosing an existing scalable numeric domain based on the properties we wish to infer, and then designing new transfer functions. Abstract interpretation provides a tool, Galois connection, that guides the systematic design of over-approximating operators and ensures (local) optimality. Only recently has this construction been adapted to the systematic design of under-approximations, in two main works. Firstly, Schmidt [18] defines Galois connections (and so best operators) for all four backward/forward over-/under-approximation cases using a higher-order powerset construction. Secondly, Massé [5] proposes an analysis parametrized by arbitrary temporal properties, including $\widetilde{\mathrm{pre}}$ operators, based on abstract domains for lower closure operators. We shy away from these higher-order constructions. We may lose optimality and generality, but achieve a more straightforward and, we believe, practical and scalable framework. Additionally, such constructions give rise to existentially quantified domains, i.e., the meaning of an abstract property is no longer "all program executions satisfy the property" but "there exists a program execution satisfying the property", so that over-approximating the property under-approximates its

meaning. Such abstractions are developed theoretically by Schmidt [22], while Goubault and Putot [23] provide a fully developed instance dedicated to abstracting computations on reals. On the contrary, we do not change the semantics of abstract elements, but only add new transfer functions, and achieve the same algorithmic complexity as forward analyses.

In [27], Cousot et al. propose a backward precondition analysis for contracts. It differs from the weakest precondition approach we follow in its treatment of non-determinism: it keeps states that, for some sequence of choices (but not necessarily all), give rise to a non-erroneous execution. Our handling of inevitability is directly inspired from the termination analysis of Cousot and Cousot [28].

Weakest liberal preconditions, introduced by Dijkstra [6], can be used to discuss about sufficient conditions. A large body of work has been devoted to their study. However, most applications to program analysis fall in the scope of deductive methods (such as [43]) based on exact reasoning using human-assisted tools. Although abstractions appear (as user-provided annotations or when the program is statically transformed into an abstract model), the automated computation is not actually performed in an abstract domain and does not use dynamic abstractions (such as widenings) to guarantee the termination and efficiency of the analysis, unlike our method .

7. Conclusion

In this article, we have presented a method to automatically infer sufficient conditions on numeric programs using abstract interpretation. The method takes into account two key aspects of sufficient conditions: the handling of non-determinism and the need to use under-approximations. Our construction is based on a systematic derivation of a backward sufficient condition semantics from a given forward invariant semantics, employing a backward "dualization" operator. By analyzing the properties of this operator, we showed how to construct the semantics and abstract it in a modular way. We then presented under-approximating abstract operators, including a lower widening to accelerate the analysis of loops, for three classic numeric domains: intervals, octagons, and polyhedra.

Despite our proof-of-concept implementation and early experimental results on very small examples, our construction and results are very preliminary and remain mostly untried. Our hope with this article is mainly to convince the reader that this constitutes a fruitful avenue of research.

Future work include the design of improved abstract operators, in particular guards (which cannot be exactly abstracted and currently rely on improvable heuristics) as well as widenings (a key adjustment variable in all abstract interpretation based analyses). We also wish to explore the design of new numeric domains that are better adapted to the inference of sufficient conditions, and study the use of partial disjunctive completions. While, on the concrete level, our semantics can be used to infer inevitability conditions and conditions on the inputs from the environment, we only experimented our abstractions on the inference of sufficient initial conditions for programs to be correct. We wish to experiment on the first two applications as well, which will surely uncover the need for new abstract domains (for instance, to represent non-uniform conditions on inputs, or to represent ranking functions that are useful to infer inevitability properties). Our long-term goal is the design of a sufficient condition generator tool for large-scale programs with a realistic semantics and apply it to the automatic generation of counter-examples.

Appendix A. Proofs

Appendix A.1. Proof of Theorem 1

PROOF. We prove: (1) $\forall T, X : \mathrm{inv}(\mathrm{cond}(T)) \subseteq T$ and (2) $\mathrm{inv}(X) \subseteq T \Longrightarrow X \subseteq \mathrm{cond}(T)$.

This is actually a special case and a consequence of the properties from Theorems 2 and 3, that are proved below. Indeed, by definition, $\mathrm{inv}(I) = \mathrm{lfp}_I\, \lambda X.X \cup \mathrm{post}(X)$. By Theorem 3.11, $\overleftarrow{\mathrm{inv}}(T) = \mathrm{gfp}_T\, \lambda X.X \cap \overleftarrow{\mathrm{post}}(X)$. We will also prove with Theorem 2 that $\overleftarrow{\mathrm{post}} = \widetilde{\mathrm{pre}}$, so that $\overleftarrow{\mathrm{inv}}(T) = \mathrm{gfp}_T\, \lambda X.X \cap \widetilde{\mathrm{pre}}(X) = \mathrm{cond}(T)$. We will also prove as a side-effect of Theorem 3.11 that inv is a $\cup$-morphism. By Theorem 2.6, $(\mathrm{inv}, \mathrm{cond})$ then forms a Galois connection and we have $\mathrm{inv}(X) \subseteq T \iff X \subseteq \mathrm{cond}(T)$. This implies immediately (2). By setting $X = \mathrm{cond}(T)$, this also implies (1). □

Appendix A.2. Proof of Theorem 2

PROOF.

1. We prove: $\overleftarrow{f}$ is monotonic and a $\cap$-morphism.
   If $A \subseteq B$, then $f(\{a\}) \subseteq A$ implies $f(\{a\}) \subseteq B$, and so, $\overleftarrow{f}(A) \subseteq \overleftarrow{f}(B)$, which proves the monotony. Moreover, $\overleftarrow{f}(\cap_{i\in I} B_i) = \{\, a \mid f(\{a\}) \subseteq \cap_{i\in I} B_i \,\} = \{\, a \mid \wedge_{i\in I}\, f(\{a\}) \subseteq B_i \,\} = \cap_{i\in I} \{\, a \mid f(\{a\}) \subseteq B_i \,\} = \cap_{i\in I} \overleftarrow{f}(B_i)$, and so, $\overleftarrow{f}$ is a $\cap$-morphism.

2. We prove: $\overleftarrow{f}$ is a sup-$\cup$-morphism.
   The sup-$\cup$-morphism property is a consequence of the monotony of $\overleftarrow{f}$: $\forall i \in I : B_i \subseteq \cup_{j\in I} B_j$, so, $\overleftarrow{f}(B_i) \subseteq \overleftarrow{f}(\cup_{j\in I} B_j)$ and $\cup_{i\in I} \overleftarrow{f}(B_i) \subseteq \overleftarrow{f}(\cup_{j\in I} B_j)$.
   To prove that $\overleftarrow{f}$ is not necessarily a $\cup$-morphism, consider the $\cup$-morphism f such that $f(\{a\}) = \{x, y\}$. Then, $\overleftarrow{f}(\{x\}) = \overleftarrow{f}(\{y\}) = \emptyset$ but $\overleftarrow{f}(\{x, y\}) = \{a\} \supsetneq \emptyset = \overleftarrow{f}(\{x\}) \cup \overleftarrow{f}(\{y\})$.
   To prove that $\overleftarrow{f}$ is not necessarily strict, consider the $\cup$-morphism f such that $f(\{a\}) = \emptyset$. Then, $\overleftarrow{f}(\emptyset) = \{a\} \supsetneq \emptyset$.

3. We prove: If f is monotonic, then $\overleftarrow{f} \circ f$ is extensive.

$$\begin{array}{lll} & a \in (\overleftarrow{f} \circ f)(A) & \\ \iff & f(\{a\}) \subseteq f(A) & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \impliedby & a \in A. & \wr \textit{ monotony of } f \int \end{array}$$

   The equivalence on the last line does not always hold, i.e., $\overleftarrow{f} \circ f$ is generally not the identity. As counter-example, consider a monotonic f such that $f(\{a\}) = f(\{b\})$. Then, $(\overleftarrow{f} \circ f)(\{a\}) = \{a, b\} \supsetneq \{a\}$.

4. We prove: If f is a $\cup$-morphism, then $f \circ \overleftarrow{f}$ is reductive.

$$\begin{array}{lll} & b \in (f \circ \overleftarrow{f})(B) & \\ \iff & \exists a \in \overleftarrow{f}(B),\, b \in f(\{a\}) & \wr \cup\textit{-morphism} \int \\ \iff & \exists a,\, f(\{a\}) \subseteq B,\, b \in f(\{a\}) & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \implies & b \in B. & \end{array}$$

   The equivalence on the last line does not always hold, i.e., $f \circ \overleftarrow{f}$ is generally not the identity. As counter-example, consider a $\cup$-morphism f such that $b \notin \mathit{Im}(f)$. Then, $(f \circ \overleftarrow{f})(\{b\}) = f(\emptyset) = \emptyset \subsetneq \{b\}$.

5. We prove: If f is extensive, then $\overleftarrow{f}$ is reductive.

$$\begin{array}{lll} & a \in \overleftarrow{f}(B) & \\ \iff & f(\{a\}) \subseteq B & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \implies & a \in B. & \wr \textit{ extensivity of } f \int \end{array}$$

   If f is instead reductive, then the last implication is reversed.

6. We prove: If f is a $\cup$-morphism, then $A \subseteq \overleftarrow{f}(B) \iff f(A) \subseteq B$.

$$\begin{array}{lll} & A \subseteq \overleftarrow{f}(B) & \\ \iff & \forall a \in A : f(\{a\}) \subseteq B & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \iff & \cup_{a\in A} f(\{a\}) \subseteq B & \\ \iff & f(A) \subseteq B. & \wr \cup\textit{-morphism} \int \end{array}$$

7. The claim $\overleftarrow{\mathrm{post}} = \widetilde{\mathrm{pre}}$ stated before the theorem is almost immediate:

$$\begin{array}{lll} & \overleftarrow{\mathrm{post}}(B) & \\ = & \{\, \sigma \in \Sigma \mid \mathrm{post}(\{\sigma\}) \subseteq B \,\} & \wr \textit{ definition of } \overleftarrow{\cdot} \int \\ = & \{\, \sigma \in \Sigma \mid \forall \sigma' : (\sigma, \sigma') \in \tau \implies \sigma' \in B \,\} & \wr \textit{ definition of } \mathrm{post} \int \\ = & \widetilde{\mathrm{pre}}(B). & \wr \textit{ definition of } \widetilde{\mathrm{pre}} \int \end{array}$$

□

Appendix A.3. Proof of Theorem 3

PROOF.

1. We prove: $\overleftarrow{\lambda A.A} = \lambda B.B$.

$$\begin{array}{lll} & \overleftarrow{\lambda A.A}(B) & \\ = & \{\, a \mid \lambda x.x(\{a\}) \subseteq B \,\} & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ = & \{\, a \mid \{a\} \subseteq B \,\} & \\ = & B. & \end{array}$$

2. We prove: $\overleftarrow{f \cup g} = \overleftarrow{f} \cap \overleftarrow{g}$.

$$\begin{array}{lll} & a \in (\overleftarrow{f \cup g})(B) & \\ \iff & f(\{a\}) \cup g(\{a\}) \subseteq B & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \iff & f(\{a\}) \subseteq B \wedge g(\{a\}) \subseteq B & \\ \iff & a \in \overleftarrow{f}(B) \wedge a \in \overleftarrow{g}(B) & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \iff & a \in (\overleftarrow{f} \cap \overleftarrow{g})(B). & \end{array}$$

3. We prove: $\overleftarrow{f \cap g} \supseteq \overleftarrow{f} \cup \overleftarrow{g}$.

$$\begin{array}{lll} & a \in (\overleftarrow{f \cap g})(B) & \\ \iff & f(\{a\}) \cap g(\{a\}) \subseteq B & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \impliedby & f(\{a\}) \subseteq B \vee g(\{a\}) \subseteq B & \\ \iff & a \in \overleftarrow{f}(B) \vee a \in \overleftarrow{g}(B) & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \iff & a \in (\overleftarrow{f} \cup \overleftarrow{g})(B). & \end{array}$$

   To prove that the equality does not necessarily hold, consider f and g such that $f(\{a\}) = \{x\}$ and $g(\{a\}) = \{y\}$. Then, $(f \cap g)(\{a\}) = \emptyset$, and so, $a \in \overleftarrow{f \cap g}(\emptyset)$. However, $a \notin \overleftarrow{f}(\emptyset)$ and $a \notin \overleftarrow{g}(\emptyset)$.

4. We prove: If f is monotonic, then $\overleftarrow{f \circ g} \subseteq \overleftarrow{g} \circ \overleftarrow{f}$.

$$\begin{array}{lll} & a \in \overleftarrow{f \circ g}(B) & \\ \iff & (f \circ g)(\{a\}) \subseteq B & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \implies & \forall b \in g(\{a\}) : f(\{b\}) \subseteq B & \wr \textit{ monotony of } f \int \\ \iff & \forall b \in g(\{a\}) : b \in \overleftarrow{f}(B) & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \iff & g(\{a\}) \subseteq \overleftarrow{f}(B) & \\ \iff & a \in (\overleftarrow{g} \circ \overleftarrow{f})(B). & \wr \textit{ def. of } \overleftarrow{\cdot} \int \end{array}$$

5. We prove: If f is a $\cup$-morphism, then $\overleftarrow{f \circ g} = \overleftarrow{g} \circ \overleftarrow{f}$.
   When f is a $\cup$-morphism, it is monotonic. The proof of 4 holds and we have moreover $(\forall b \in g(\{a\}) : f(\{b\}) \subseteq B) \implies (f \circ g)(\{a\}) = \cup \{\, f(\{b\}) \mid b \in g(\{a\}) \,\} \subseteq B$, which proves the equality.

6. We prove: $f \subseteq g \Longrightarrow \overleftarrow{g} \subseteq \overleftarrow{f}$.

$$\begin{array}{lll} & a \in \overleftarrow{g}(B) & \\ \iff & g(\{a\}) \subseteq B & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \Longrightarrow & f(\{a\}) \subseteq B & \wr\, f \subseteq g \int \\ \iff & a \in \overleftarrow{f}(B). & \wr \textit{ def. of } \overleftarrow{\cdot} \int \end{array}$$

7. We prove: If f and g are $\cup$-morphisms, then $f \subseteq g \iff \overleftarrow{g} \subseteq \overleftarrow{f}$.
The $\Longrightarrow$ part has been proved in 6. Suppose instead now that $\overleftarrow{g} \subseteq \overleftarrow{f}$, then we have on singletons:

$$\begin{array}{lll} & b \in f(\{a\}) & \\ \iff & \forall B : (f(\{a\}) \subseteq B \Longrightarrow b \in B) & \\ \iff & \forall B : (a \in \overleftarrow{f}(B) \Longrightarrow b \in B) & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \Longrightarrow & \forall B : (a \in \overleftarrow{g}(B) \Longrightarrow b \in B) & \wr\, \overleftarrow{g} \subseteq \overleftarrow{f} \int \\ \iff & \forall B : (g(\{a\}) \subseteq B \Longrightarrow b \in B) & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \iff & b \in g(\{a\}). & \end{array}$$

So, $\forall a : f(\{a\}) \subseteq g(\{a\})$. When f and g are $\cup$-morphisms, the property lifts to arbitrary sets, and we have $f \subseteq g$. This property implies that $\overleftarrow{g} = \overleftarrow{f} \Longrightarrow f = g$.

8. We prove: If f and g are monotonic and $f \circ g = g \circ f = \lambda x.x$, then $\overleftarrow{f} = g$.

$$\begin{array}{lll} & a \in \overleftarrow{f}(B) & \\ \iff & f(\{a\}) \subseteq B & \wr \textit{ def. } \overleftarrow{\cdot} \int \\ \Longrightarrow & (g \circ f)(\{a\}) \subseteq g(B) & \wr \textit{ monotony of } g \int \\ \iff & a \in g(B). & \wr\, g \circ f = \lambda x.x \int \end{array}$$

Moreover:

$$\begin{array}{lll} & a \in \overleftarrow{f}(B) & \\ \iff & f(\{a\}) \subseteq B & \wr \textit{ def. } \overleftarrow{\cdot} \int \\ \iff & (f \circ g \circ f)(\{a\}) \subseteq (f \circ g)(B) & \wr\, f \circ g = \lambda x.x \int \\ \Longleftarrow & (g \circ f)(\{a\}) \subseteq g(B) & \wr \textit{ monotony of } f \int \\ \iff & a \in g(B). & \wr\, g \circ f = \lambda x.x \int \end{array}$$

9. We prove: If f is a $\cup$-morphism, x is a pre-fixpoint of f, and y is a post-fixpoint of $\overleftarrow{f}$, then $\mathrm{lfp}_x\, f \subseteq y \iff x \subseteq \mathrm{gfp}_y\, \overleftarrow{f}$.
By Cousot's constructive version of Tarski's fixpoint theorem [25], we have $\forall x : \mathrm{lfp}_x\, f = \cup_{i \in \mathbb{N}} f^i(x)$ and $\forall y : \mathrm{gfp}_y\, \overleftarrow{f} = \cap_{i \in \mathbb{N}} \overleftarrow{f}^i(y)$. Moreover, each f^i is, as f, a $\cup$-morphism, and so, $\overleftarrow{f^i} = \overleftarrow{f}^i$ by point 5.

$$\begin{array}{lll} & \mathrm{lfp}_x\, f \subseteq y & \\ \iff & \cup_{i \in \mathbb{N}} f^i(x) \subseteq y & \wr \textit{ Tarski's theorem} \int \\ \iff & \forall i \in \mathbb{N} : f^i(x) \subseteq y & \\ \iff & \forall i \in \mathbb{N} : x \subseteq \overleftarrow{f^i}(y) & \wr \textit{ Galois connection} \int \\ \iff & \forall i \in \mathbb{N} : x \subseteq \overleftarrow{f}^i(y) & \wr \cup\textit{-morphism of } f^i \int \\ \iff & x \subseteq \cap_{i \in \mathbb{N}} \overleftarrow{f}^i(y) & \wr\, \{\, \overleftarrow{f}^i(y) \mid i \in \mathbb{N} \,\} \textit{ forms a chain} \int \\ \iff & x \subseteq \mathrm{gfp}_y\, \overleftarrow{f}. & \wr \textit{ Tarski's theorem} \int \end{array}$$

10. We prove: If f is an extensive $\cup$-morphism, then $\overleftarrow{\lambda x.\mathrm{lfp}_x\, f} = \lambda y.\mathrm{gfp}_y\, \overleftarrow{f}$.
    As f is extensive, $\overleftarrow{f}$ is reductive. As a consequence, every $x \in \mathcal{P}(X)$ is a pre-fixpoint of f and a post-fixpoint of $\overleftarrow{f}$. Applying the preceding point, we get a Galois connection: $\mathcal{P}(X) \underset{\lambda x.\mathrm{lfp}_x\, f}{\overset{\lambda y.\mathrm{gfp}_y\, \overleftarrow{f}}{\leftrightarrows}} \mathcal{P}(Y)$. By Theorem 2.6, $\overleftarrow{\lambda x.\mathrm{lfp}_x\, f}$ is the (unique) Galois adjoin of $\lambda x.\mathrm{lfp}_x\, f$, hence $\lambda y.\mathrm{gfp}_y\, \overleftarrow{f} = \overleftarrow{\lambda x.\mathrm{lfp}_x\, f}$.
11. We prove: If f is a $\cup$-morphism, then $\overleftarrow{\lambda x.\mathrm{lfp}_x\, (\lambda z.z \cup f(z))} = \lambda y.\mathrm{gfp}_y(\lambda z.z \cap \overleftarrow{f}\,(z))$.
    This is a simple application of the preceding point using the function $h \overset{\text{def}}{=} \lambda z.z \cup f(z)$, which is an extensive $\cup$-morphism, and the fact that $\overleftarrow{h} = \lambda z.z \cap \overleftarrow{f}\,(z)$ by properties 1 and 2.
    We note in passing that $\lambda x.\mathrm{lfp}_x\, h$ is a $\cup$-morphism, as the infinite join of $\cup$-morphisms. This property is useful when using $\lambda x.\mathrm{lfp}_x\, h$ as argument in the above properties that require it.

□

Appendix A.4. Proof of Theorem 4

PROOF. We prove: $\gamma(\lim_{X^\sharp} \lambda Y^\sharp.Y^\sharp \mathbin{\underline{\triangledown}} F^\sharp(Y^\sharp)) \subseteq \mathrm{gfp}_X\, F$.
Let us note $Z^\sharp \overset{\text{def}}{=} \lim_{X^\sharp} \lambda Y^\sharp.Y^\sharp \mathbin{\underline{\triangledown}} F^\sharp(Y^\sharp)$. Then:

$$\begin{array}{lll} & F(\gamma(Z^\sharp)) & \\ \supseteq & \gamma(F^\sharp(Z^\sharp)) & \wr \textit{ soundness of } F^\sharp \int \\ \supseteq & \gamma(Z^\sharp \mathbin{\underline{\triangledown}} F^\sharp(Z^\sharp)) & \wr \textit{ soundness of } \underline{\triangledown} \int \\ = & \gamma(Z^\sharp) & \wr \textit{ definition of } \lim \int \end{array}$$

hence, $\gamma(Z^\sharp)$ is a pre-fixpoint of F, which is moreover smaller than $\gamma(X^\sharp)$ by soundness of $\underline{\triangledown}$, and so, smaller than X by soundness of $X^\sharp$. This implies [25] that $\gamma(Z^\sharp)$ is smaller than $\mathrm{gfp}_X\, F$. □

Appendix A.5. Proof of Theorem 6

PROOF. We prove: $\gamma_c(C \setminus \{\vec{a} \cdot \vec{x} \geq b\}) \subseteq \overleftarrow{\tau}\{\!|\, \vec{a} \cdot \vec{x} \geq b \,|\!\}\, \gamma_c(C)$.
Take $\rho \in \gamma_c(C \setminus \{\vec{a} \cdot \vec{x} \geq b\})$. Then, either $\vec{a} \cdot \vec{\rho} \geq b$, in which case $\rho \in \gamma_c(C \cup \{\vec{a} \cdot \vec{x} \geq b\}) \subseteq \gamma_c(C)$, or $\vec{a} \cdot \vec{\rho} < b$. In both cases, $\rho \in \gamma_c(C) \cup \{\, \rho \mid \vec{a} \cdot \vec{\rho} < b \,\} = \overleftarrow{\tau}\{\!|\, \vec{a} \cdot \vec{x} \geq b \,|\!\}\, \gamma_c(C)$.

We now justify the correctness of the two heuristics we proposed in Sec. 3.2, i.e., pre-processing the argument P: (1) adding the constraint $\vec{a} \cdot \vec{x} \geq b$, and (2) adding rays $\vec{r}$ that satisfy $\tau\{\!|\, \vec{a} \cdot \vec{x} \geq b \,|\!\}\, \gamma_g(V_P, R_p \cup \{\vec{r}\}) = \tau\{\!|\, \vec{a} \cdot \vec{x} \geq b \,|\!\}\, \gamma_g(V_P, R_P)$. Adding a constraint (1) is always sound as we under-approximate the argument which, by monotony of the concrete backward operator, under-approximates the result. When adding rays (2), we took care to only add environments ρ such that $\vec{a} \cdot \vec{\rho} < b$, which are added by the concrete function $\overleftarrow{\tau}\{\!|\, \vec{a} \cdot \vec{x} \geq b \,|\!\}$ anyway, so the transformation is sound. □

Appendix A.6. Proof of Theorem 7

PROOF. We simply observe that:

- $\tau\{\!|\, \vec{a} \cdot \vec{x} > b \,|\!\} \subseteq \tau\{\!|\, \vec{a} \cdot \vec{x} \geq b \,|\!\}$,
- $\tau\{\!|\, \vec{a} \cdot \vec{x} \geq [b; c] \,|\!\} = \tau\{\!|\, \vec{a} \cdot \vec{x} \geq b \,|\!\}$,
- $\tau\{\!|\, \vec{a} \cdot \vec{x} = [b; c] \,|\!\} = \tau\{\!|\, \vec{a} \cdot \vec{x} \geq b \,|\!\} \circ \tau\{\!|\, \vec{a} \cdot \vec{x} \leq c \,|\!\}$.

and apply Theorem 3 and Theorem 6. □

Appendix A.7. Proof of Theorem 8

PROOF. We prove: If R is closed and convex, then $\overleftarrow{\tau}\{\!|\, V := [-\infty;+\infty] \,|\!\}\, R$ is either R or $\emptyset$.
Let us note $R' \stackrel{\text{def}}{=} \overleftarrow{\tau}\{\!|\, V := [-\infty;+\infty] \,|\!\}\, R = \{\, \rho \in \mathcal{E} \mid \forall v \in \mathbb{R} : \rho[V \mapsto v] \in R \,\}$. We note that $R' \subseteq R$ (by choosing $v = \rho(V)$, we get $\rho \in R$).

We now prove that, if $R' \neq \emptyset$, then $R' = R$. Assume that $R' \neq \emptyset$ and, *ad absurdum*, that $R' \subsetneq R$, i.e., there exist $\rho, \rho' \in R$ such that $\rho \in R'$ but $\rho' \notin R'$. Thus, $\exists v' \in \mathbb{R} : \rho'[V \mapsto v'] \notin R$. For any $\epsilon \in (0,1]$, we now construct a point ρ'_ϵ in R that is at distance less than ϵ from $\rho'[V \mapsto v']$. We take ρ'_ϵ on the segment between $\rho' \in R$ and $\rho[V \mapsto M_\epsilon] \in R$: $\rho'_\epsilon \stackrel{\text{def}}{=} (1-\alpha_\epsilon)\rho' + \alpha_\epsilon \rho[V \mapsto M_\epsilon]$, for some well-chosen M_ϵ and α_ϵ. More precisely, we choose $\alpha_\epsilon = \epsilon / \max\{\,1, |\rho(W) - \rho'(W)| \mid W \neq V\,\}$ and $M_\epsilon = \rho'(V) + (v' - \rho'(V))/\alpha_\epsilon$. This implies $\forall W \neq V : |\rho'_\epsilon(W) - \rho'[V \mapsto v'](W)| = |((1-\alpha_\epsilon)\rho'(W) + \alpha_\epsilon\rho(W)) - \rho'(W)| = \alpha_\epsilon |\rho(W) - \rho'(W)| \leq \epsilon$. Moreover $|\rho'_\epsilon(V) - \rho'[V \mapsto v'](V)| = |((1-\alpha_\epsilon)\rho'(V) + \alpha_\epsilon M_\epsilon) - v'| = |\rho'(V) - \alpha_\epsilon\rho'(V) + \alpha_\epsilon\rho'(V) + v' - \rho'(V) - v'| = 0$. So, we indeed have $|\rho'_\epsilon - \rho'[V \mapsto v']|_\infty \leq \epsilon$. Finally, by convexity of R, we have $\rho'_\epsilon \in R$. We can thus construct a sequence of points in R that converges to $\rho'[V \mapsto v']$. As R is closed, this implies $\rho'[V \mapsto v'] \in R$, and so, our hypothesis $\rho' \notin R'$ is false.

We now prove: $\tau\{\!|\, V := [-\infty;+\infty] \,|\!\}\, R = R \iff R = \overleftarrow{\tau}\{\!|\, V := [-\infty;+\infty] \,|\!\}\, R$.
We first apply Theorem 2.6, to get $R \subseteq \overleftarrow{\tau}\{\!|\, V := [-\infty;+\infty] \,|\!\}\, R \iff \tau\{\!|\, V := [-\infty;+\infty] \,|\!\}\, R \subseteq R$. The proof is completed by using the fact that $R \subseteq \tau\{\!|\, V := [-\infty;+\infty] \,|\!\}\, R$, and $\overleftarrow{\tau}\{\!|\, V := [-\infty;+\infty] \,|\!\}\, R \subseteq R$ which we just proved. □

Appendix A.8. Proof of Theorem 9

PROOF.

1. We prove: If R is convex, then $\overleftarrow{\tau}\{\!|\, V := V + [a;b] \,|\!\}\, R = \tau\{\!|\, V := V - a \,|\!\}\, R \cap \tau\{\!|\, V := V - b \,|\!\}\, R$.

$$\begin{aligned} & \overleftarrow{\tau}\{\!|\, V := V + [a;b] \,|\!\}\, R \\ = \; & \{\, \rho \mid \forall x \in [a;b] : \rho[V \mapsto \rho(V) + x] \in R \,\} \\ = \; & \{\, \rho \mid \forall x \in \{a,b\} : \rho[V \mapsto \rho(V) + x] \in R \,\} \\ = \; & \{\, \rho \mid \rho[V \mapsto \rho(V) + a] \in R \,\} \cap \{\, \rho \mid \rho[V \mapsto \rho(V) + b] \in R \,\} \\ = \; & \tau\{\!|\, V := V - a \,|\!\}\, R \cap \tau\{\!|\, V := V - b \,|\!\}\, R. \end{aligned}$$

   The fact that $\forall x \in [a;b] : \rho[V \mapsto \rho(V) + x] \in R \iff \forall x \in \{a,b\} : \rho[V \mapsto \rho(V) + x] \in R$ comes form the hypothesis that R is convex.

2. We prove: If R is convex, then: $\overleftarrow{\tau}\{\!|\, V := [a;b] \,|\!\}\, R =$
   $(\tau\{\!|\, V := [-\infty;+\infty] \,|\!\} \circ (\tau\{\!|\, V := V - a \,|\!\} \cap \tau\{\!|\, V := V - b \,|\!\}) \circ \tau\{\!|\, V \geq a \wedge V \leq b \,|\!\})R$.
   By definition $\overleftarrow{\tau}\{\!|\, V := [a;b] \,|\!\}\, R = \{\, \rho \in \mathcal{E} \mid \forall y \in [a;b] : \rho[V \mapsto y] \in R \,\}$.
   Assume that $\rho \in (\tau\{\!|\, V := [-\infty;+\infty] \,|\!\} \circ (\tau\{\!|\, V := V - a \,|\!\} \cap \tau\{\!|\, V := V - b \,|\!\}) \circ \tau\{\!|\, V \geq a \wedge V \leq b \,|\!\})R$.
   By definition of $\tau\{\!|\, V := [-\infty;+\infty] \,|\!\}$, there exists some x such that:

$$\rho[V \mapsto x] \in ((\tau\{\!|\, V := V - a \,|\!\} \cap \tau\{\!|\, V := V - b \,|\!\}) \circ \tau\{\!|\, V \geq a \wedge V \leq b \,|\!\})R \;.$$

   We have $\rho[V \mapsto x] \in (\tau\{\!|\, V := V - a \,|\!\} \circ \tau\{\!|\, V \geq a \wedge V \leq b \,|\!\})R$.
   Hence $\rho[V \mapsto x + a] \in \tau\{\!|\, V \geq a \wedge V \leq b \,|\!\}\, R$. This implies in particular $x \geq 0$. Likewise, we have $\rho[V \mapsto x + b] \in \tau\{\!|\, V \geq a \wedge V \leq b \,|\!\}\, R$, and so $x \leq 0$. We deduce that $x = 0$ and $\rho[V \mapsto a]$, $\rho[V \mapsto b] \in \tau\{\!|\, V \geq a \wedge V \leq b \,|\!\}\, R \subseteq R$. By convexity of R, $\forall y \in [a;b] : \rho[V \mapsto y] \in R$. Hence, $\rho \in \overleftarrow{\tau}\{\!|\, V := [a;b] \,|\!\}\, R$.
   Conversely, suppose that $\rho \in \overleftarrow{\tau}\{\!|\, V := [a;b] \,|\!\}\, R$. Then, by definition, $\rho[V \mapsto a]$, $\rho[V \mapsto b] \in R$. Obviously, $\rho[V \mapsto a]$, $\rho[V \mapsto b] \in \tau\{\!|\, V \geq a \wedge V \leq b \,|\!\}\, R$.
   As a consequence $\rho[V \mapsto 0] \in ((\tau\{\!|\, V := V - a \,|\!\} \cap \tau\{\!|\, V := V - b \,|\!\}) \circ \tau\{\!|\, V \geq a \wedge V \leq b \,|\!\})R$. We deduce that $\rho \in (\tau\{\!|\, V := [-\infty;+\infty] \,|\!\} \circ (\tau\{\!|\, V := V - a \,|\!\} \cap \tau\{\!|\, V := V - b \,|\!\}) \circ \tau\{\!|\, V \geq a \wedge V \leq b \,|\!\})R$.
   Note that we can safely under-approximate $\overleftarrow{\tau}\{\!|\, V := X \,|\!\}$ for any set X as $\overleftarrow{\tau}\{\!|\, V := [\min X; \max X] \,|\!\}$.

3. We prove: $\overleftarrow{\tau}\{\!|\, V := W \,|\!\}= \tau\{\!|\, V := [-\infty;+\infty] \,|\!\} \circ \tau\{\!|\, V = W \,|\!\}$.
   By definition, $\overleftarrow{\tau}\{\!|\, V := W \,|\!\} R = \{\, \rho \in \mathcal{E} \mid \rho[V \mapsto \rho(W)] \in R \,\}$.
   Take $\rho \in (\tau\{\!|\, V := [-\infty;+\infty] \,|\!\} \circ \tau\{\!|\, V = W \,|\!\})R$. Then, by definition, $\exists x : \rho[V \mapsto x] \in \tau\{\!|\, V = W \,|\!\} R$, i.e., $\rho[V \mapsto x] \in R$ and $\rho(W) = \rho[V \mapsto x](V) = x$. We deduce that $\rho[V \mapsto \rho(W)] = \rho[V \mapsto x] \in R$, and so, $\rho \in \overleftarrow{\tau}\{\!|\, V := W \,|\!\} R$.
   Take now $\rho \in \overleftarrow{\tau}\{\!|\, V := W \,|\!\} R$. Then, $\rho[V \mapsto \rho(W)] \in R$. As obviously $\rho[V \mapsto \rho(W)]$ satisfies the constraint $V = W$, we have $\rho[V \mapsto \rho(W)] \in \tau\{\!|\, V = W \,|\!\} R$. Then, $\rho \in (\tau\{\!|\, V := [-\infty;+\infty] \,|\!\} \circ \tau\{\!|\, V = W \,|\!\})R$.

□

Appendix A.9. Proof of Theorem 11

PROOF. We prove: $\underline{\triangledown}$ is a lower widening.
We first prove that $A \,\underline{\triangledown}\, B \subseteq A \cap B$, i.e., using the generator representation $\gamma_g(V_{A\underline{\triangledown}B}, R_{A\underline{\triangledown}B}) \subseteq \gamma_g(V_A, R_A) \cap \gamma_g(V_B, R_B)$. We first note that $V_{A\underline{\triangledown}B} \subseteq V_A$ and $R_{A\underline{\triangledown}B} \subseteq R_A$, which implies directly $\gamma_g(V_{A\underline{\triangledown}B}, R_{A\underline{\triangledown}B}) \subseteq \gamma_g(V_A, R_A)$. Consider now a point $\vec{x} \in \gamma_g(V_{A\underline{\triangledown}B}, R_{A\underline{\triangledown}B})$. By definition, $\vec{x}$ can be written as $\vec{x} = \sum_{\vec{v}\in V_{A\underline{\triangledown}B}} \alpha_{\vec{v}}\vec{v} + \sum_{\vec{r}\in R_{A\underline{\triangledown}B}} \beta_{\vec{r}}\vec{r}$ for some $\alpha_{\vec{v}}, \beta_{\vec{r}} \geq 0$ such that $\sum_{\vec{v}} \alpha_{\vec{v}} = 1$. As $V_{A\underline{\triangledown}B} \subseteq B$, by convexity of B, $\sum_{\vec{v}\in V_{A\underline{\triangledown}B}} \alpha_{\vec{v}}\vec{v} \in B$. Moreover, for each $\vec{r} \in R_{A\underline{\triangledown}B}$, we have $B \oplus \beta_{\vec{r}}\vec{r} \subseteq B \oplus \mathbb{R}^+\vec{r} = B$, so, $\vec{x} \in B$.

The termination follows simply from the fact that, in any sequence $X_{i+1} = X_i \,\underline{\triangledown}\, Y_{i+1}$, the set of generators in X_i decreases. Whenever $V_{X_i} = \emptyset$, the corresponding polyhedron is empty. □

Appendix A.10. Proof of Theorem 12

PROOF. We prove: $\underline{\triangledown}$ is a lower widening.
We obviously have $[a;b] \,\underline{\triangledown}\, [c;d] \subseteq [a;b] \cap [c;d]$. We now prove termination. Consider *ad absurdum* an infinite sequence $[a_{i+1};b_{i+1}] \stackrel{\text{def}}{=} [a_i;b_i] \,\underline{\triangledown}\, [c_{i+1};d_{i+1}]$. Then, we always have $a_i < b_i$ as a widening sequence necessarily terminates when reaching a singleton. Consider now L_i, H_i as defined in (18). Then, as a_i is increasing and b_i is decreasing, $|L_i|$ and $|H_i|$ are both decreasing. The fact that the sequence does not terminate implies that $\forall i : L_i, H_i \neq \emptyset$, $\min L_i \leq \max H_i$. Hence, at some point $|L_{n+1}| = |L_n|$ and $|H_{n+1}| = |H_n|$, which implies $a_{n+1} = a_n$ and $b_{n+1} = b_n$, and the sequence is in fact stable. □

Appendix A.11. Proof of Theorem 13

PROOF.

1. We prove: If $e \sqsubseteq_D f$, then $\overleftarrow{\tau}\{\!|\, V := e \,|\!\} R \supseteq (\overleftarrow{\tau}\{\!|\, V := f \,|\!\} R) \cap D$.

$$\begin{array}{lll} & \overleftarrow{\tau}\{\!|\, V := e \,|\!\} R & \\ \supseteq & (\overleftarrow{\tau}\{\!|\, V := e \,|\!\} R) \cap D & \\ = & \{\, \rho \in D \mid \forall v \in [\![\, e \,]\!]\rho : \rho[V \mapsto v] \in R \,\} & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \supseteq & \{\, \rho \in D \mid \forall v \in [\![\, f \,]\!]\rho : \rho[V \mapsto v] \in R \,\} & \wr\, e \sqsubseteq_D f \,\int \\ = & (\overleftarrow{\tau}\{\!|\, V := f \,|\!\} R) \cap D. & \end{array}$$

2. We prove: If $e \sqsubseteq_D f$, then $\overleftarrow{\tau}\{\!|\, e \,|\!\} R \supseteq (\overleftarrow{\tau}\{\!|\, f \,|\!\} R) \cap D$.

$$\begin{array}{lll} & \overleftarrow{\tau}\{\!|\, e \,|\!\} R & \\ = & R \cup \{\, \rho \in \mathcal{E} \mid [\![\, e \,]\!]\rho = \{f\} \,\} & \wr \textit{ def. of } \overleftarrow{\cdot} \int \\ \supseteq & R \cup \{\, \rho \in D \mid [\![\, e \,]\!]\rho = \{f\} \,\} & \\ \supseteq & R \cup \{\, \rho \in D \mid [\![\, f \,]\!]\rho = \{f\} \,\} & \wr\, e \sqsubseteq_D f \,\int \\ \supseteq & D \cap (R \cup \{\, \rho \mid [\![\, f \,]\!]\rho = \{f\} \,\}) & \\ = & D \cap (\overleftarrow{\tau}\{\!|\, f \,|\!\} R). & \wr \textit{ def. of } \overleftarrow{\cdot} \int \end{array}$$

□

Appendix A.12. Proof of Theorem 16

PROOF.

1. We prove: If $(\sigma_0, l) \in \mathrm{cond}(T')$, then all the traces starting in σ_0 eventually enter a state in T.
   Let us note $S' \stackrel{\text{def}}{=} \mathrm{cond}(T')$ in the transition system (Σ', τ'). Consider a state $(\sigma_0, l) \in S'$ and a maximal trace $t = \sigma_0, \ldots, \sigma_i, \ldots$ for (Σ, τ) starting in state σ_0. As we assumed that (Σ, τ) has no blocking state, the maximal trace t is infinite. We note $m \stackrel{\text{def}}{=} \min(l, \max\{\, i \mid \forall j < i : \sigma_j \notin T \,\})$. As $m \leq l$, m is finite. We construct the trace t' in (Σ', τ') as follows: $t' \stackrel{\text{def}}{=} (\sigma_0, l), (\sigma_1, l-1), \ldots, (\sigma_m, l-m)$. Then, t' obeys τ'. As moreover, $(\sigma_0, l) \in S' = \mathrm{cond}(T')$, we have $\forall i : (\sigma_i, l_i) \in T'$. If $m = l$, this gives $l - m = 0$, and so $\sigma_m \in T$ by definition of T'. If $m < l$, then $m = \max\{\, i \mid \forall j < i : \sigma_j \notin T \,\})$, which implies $\sigma_m \in T$. We note that all the traces starting in state σ_0 reach a state in T in l steps or less.
2. We prove: If the non-determinism in τ is finite, and all the traces starting in σ eventually enter state T, then $\exists l : (\sigma, l) \in \mathrm{cond}(T')$.
   Consider a state $\sigma \in \Sigma$ such that all the traces in (Σ, τ) from σ eventually reach a state in T. For each maximal such trace, we consider its finite prefix until it reaches T, i.e., $t = \sigma_1, \ldots, \sigma_{|t|}$ such that $\sigma_1 = \sigma$, $\sigma_{|t|} \in T$, and $i < |t| \implies \sigma_i \notin T$. The set of all these finite trace prefixes forms a tree rooted at σ. By hypothesis, this tree has no infinite path. We now use the extra hypothesis that the non-determinism in τ is finite, which ensures that the tree is finitely branching. By the contrapositive of König's lemma, the tree is then finite. We denote by l its (finite) depth.
   We now argue that $(\sigma, l) \in S' \stackrel{\text{def}}{=} \mathrm{cond}(T')$.
   It is sufficient to prove that, for each finite trace $t' \stackrel{\text{def}}{=} (\sigma_0, l_0), \ldots, (\sigma_n, l_n)$ for (Σ', τ') starting in state $(\sigma_0, l_0) \stackrel{\text{def}}{=} (\sigma, l)$, we have $\forall i : (\sigma_i, l_i) \in T'$. By definition of τ', $l_i = l - i$. Thus, if $n < l$, then $\forall i : l_i > 0$ and the property is obvious. Otherwise, $n \geq l$, and it is sufficient to prove that $\exists i : \sigma_i \in T$. Suppose that this is not the case and $\forall i : \sigma_i \notin T$. Then, by definition of τ', the trace $t = \sigma_0, \ldots, \sigma_l$ in Σ obeys τ. We have constructed a trace for τ starting in σ with length strictly greater than l that does not encounter a state in T, which contradicts the definition of l.

□

References

[1] A. Miné, Inferring sufficient conditions with backward polyhedral under-approximations, in: Proc. of the 4th Int. Workshop on Numerical and Symbolic Abstract Domains (NSAD'12), Vol. 287 of Electron. Notes Theor. Comput. Sci., Elsevier, 2012, pp. 89–100.
[2] K. Arnout, B. Meyer, Uncovering hidden contracts: The .NET example, IEEE Computer 36 (11) (2003) 48–55.
[3] J. Brauer, A. Simon, Inferring definite counterexamples through under-approximation, in: NASA Formal Methods, Vol. 7226 of Lect. Notes Comput. Sci, 2012, pp. 54–69.
[4] C. Popeea, D. N. Xu, W.-N. Chin, A practical and precise inference and specializer for array bound checks elimination, in: Proc. of the ACM SIGPLAN Symp. on Partial Evaluation and Semantics-based Program Manipulation (PEPM'08), ACM, 2008, pp. 177–187.
[5] D. Massé, Temporal property-driven verification by abstract interpretation, Ph.D. thesis, École Polytechnique, Palaiseau, France (Dec. 2002).
[6] E. W. Dijkstra, Guarded commands, non-determinacy and formal derivation of programs, Comm. of the ACM 18 (8) (1975) 453–457.
[7] J. Morris, Non-deterministic expressions and predicate transformers, Information Processing Letters 61 (1997) 241–246.
[8] J.-C. Filliâtre, Deductive software verification, Int. Journal on Software Tools for Technology Transfer (STTT) 13 (5) (2011) 397–403.
[9] E. A. Emerson, The beginning of model checking: A personal perspective, in: 25 Years of Model Checking, Springer, 2008, pp. 27–45.
[10] A. Armando, J. Mantovani, L. Platania, Bounded model checking of software using SMT solvers instead of SAT solvers, Int. J. Softw. Tools Technol. Transf. 11 (1) (2009) 69–83.
[11] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement for symbolic model checking, J. ACM 50 (5) (2003) 752–794.
[12] P. Cousot, Méthodes itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes (in French), Ph.D. thesis, Université scientifique et médicale de Grenoble, Grenoble, France (21 Mar. 1978).
[13] J. Bertrane, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, X. Rival, Static analysis and verification of aerospace software by abstract interpretation, in: AIAA Infotech@Aerospace, no. 2010-3385, AIAA, 2010, pp. 1–38.
[14] P. Cousot, R. Cousot, Static determination of dynamic properties of programs, in: Proc. of the Second Int. Symp. on Programming, Dunod, Paris, France, 1976, pp. 106–130.
[15] P. Cousot, N. Halbwachs, Automatic discovery of linear restraints among variables of a program, in: Proc. of the 5th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'78), ACM Press, 1978, pp. 84–97.

[16] A. Miné, The octagon abstract domain, Higher-Order and Symbolic Computation 19 (1) (2006) 31–100.
[17] F. Bourdoncle, Abstract debugging of higher-order imperative languages, in: Proc. of the ACM Conf. on Programing Language Design and Implementation (PLDI'93), ACM, 1993, pp. 46–55.
[18] D. A. Schmidt, Closed and logical relations for over- and under-approximation of powersets, in: Proc. of the 11th Int. Symp. on Static Analysis (SAS'04), Vol. 3148 of Lect. Notes Comput. Sci, Springer, 2004, pp. 22–37.
[19] Y. Moy, Sufficient preconditions for modular assertion checking, in: Proc. of the 9th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'08), Vol. 4905 of Lect. Notes Comput. Sci, Springer, 2008, pp. 188–202.
[20] J. C. King, Symbolic execution and program testing, Comm. of the ACM 19 (7) (1976) 385–394.
[21] T. Lev-Ami, M. Sagiv, T. Reps, S. Gulwani, Backward analysis for inferring quantified pre-conditions, Tech. Rep. TR-2007-12-01, Tel Aviv University (Dec. 2007).
[22] D. A. Schmidt, Underapproximating predicate transformers, in: Proc. of 13th Int. Symp. on Static Analysis (SAS'06), Vol. 4134 of Lect. Notes Comput. Sci, Springer, 2006, pp. 127–143.
[23] E. Goubault, S. Putot, Under-approximations of computations in real numbers based on generalized affine arithmetic, in: Proc. of the 14th Int. Symp. on Static Analysis (SAS'07), Vol. 4634 of Lect. Notes Comput. Sci, Springer, 2007, pp. 137–152.
[24] A. Tarski, A lattice theoretical fixpoint theorem and its applications, Pacific Journal of Mathematics 5 (1955) 285–310.
[25] P. Cousot, R. Cousot, Constructive versions of Tarski's fixed point theorems, Pacific Journal of Mathematics 81 (1) (1979) 43–57.
[26] X. Rival, Understanding the origin of alarms in Astrée, in: Proc. of the 12th Int. Symp. on Static Analysis (SAS'05), Vol. 3672 of Lect. Notes Comput. Sci, Springer, 2005, pp. 303–319.
[27] P. Cousot, R. Cousot, F. Logozzo, Precondition inference from intermittent assertions and application to contracts on collections, in: Proc. of the 12th Int. Conf. on Verification, Model Checking and Abstract Interpretation (VMCAI'11), Vol. 6538 of Lect. Notes Comput. Sci, Springer, 2011, pp. 150–168.
[28] P. Cousot, R. Cousot, An abstract interpretation framework for termination, in: Conf. Rec. of the 39^{th} Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, ACM Press, New York, Philadelphia, PA, 2012, pp. 245–258.
[29] G. Lalire, M. Argoud, B. Jeannet, Interproc static analyzer, `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi` (2011).
[30] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Proc. of the 4th ACM Symp. on Principles of Programming Languages (POPL'77), ACM, 1977, pp. 238–252.
[31] D. A. Schmidt, Abstract interpretation from a denotational semantics perspective, in: Proc. 25th Conf. Mathematical Foundations of Programming Semantics, Vol. 249 of Electron. Notes Theor. Comput. Sci., Elsevier, 2009, pp. 19–37.
[32] F. Bourdoncle, Efficient chaotic iteration strategies with widenings, in: Proc. of the Int. Conf. on Formal Methods in Programming and their Applications (FMPA'93), Vol. 735 of Lect. Notes Comput. Sci, Springer, 1993, pp. 128–141.
[33] D. Nguyen Que, Robust and generic abstract domain for static program analysis: the polyhedral case, Ph.D. thesis, École des Mines de Paris (2010).
[34] K. Larsen, F. Larsson, P. Pettersson, W. Yi, Efficient verification of real-time systems: Compact data structure and state-space reduction, in: IEEE RTSS'97, IEEE CS Press, 1997, pp. 14–24.
[35] P. Cousot, R. Cousot, Comparing the Galois connection and widening/narrowing approaches to abstract interpretation, invited paper, in: Proc. of the Int. Workshop on Programming Language Implementation and Logic Programming (PLILP'92), Vol. 631 of Lect. Notes Comput. Sci, Springer, 1992, pp. 269–295.
[36] R. Bagnara, P. M. Hill, E. Ricci, E. Zaffanella, Precise widening operators for convex polyhedra, Science of Computer Programming 58 (1–2) (2005) 28–56.
[37] R. Bagnara, P. M. Hill, E. Zaffanella, Weakly-relational shapes for numeric abstractions: Improved algorithms and proofs of correctness, Formal Methods in System Design 35 (3) (2009) 279–323.
[38] A. Miné, Symbolic methods to enhance the precision of numerical abstract domains, in: Proc. of the 7th Int. Conf. on Verification, Model Checking, and Abstract Interpretation (VMCAI'06), Vol. 3855 of Lect. Notes Comput. Sci, Springer, 2006, pp. 348–363.
[39] B. Jeannet, A. Miné, Apron: A library of numerical abstract domains for static analysis, in: Proc. of the 21th Int. Conf. on Computer Aided Verification (CAV'09), Vol. 5643 of Lect. Notes Comput. Sci, Springer, 2009, pp. 661–667.
[40] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Conf. Rec. of the Sixth Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages (POPL'79), ACM Press, 1979, pp. 269–282.
[41] C. Urban, The abstract domain of segmented ranking functions, in: Proc. of the 20th Int. Symp. on Static Analysis (SAS'13), Vol. 7935 of Lect. Notes Comput. Sci, Springer, 2013, pp. 43–62.
[42] A. Miné, The Banal static analyzer prototype, `http://www.di.ens.fr/~mine/banal` (2012).
[43] C. Flanagan, R. Leino, M. Lillibridge, G. Nelson, J. Saxe, R. Stata, Extended static checking for Java, in: Proc. of the SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'02), ACM, 2002, pp. 234–245.